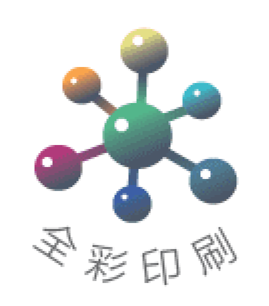
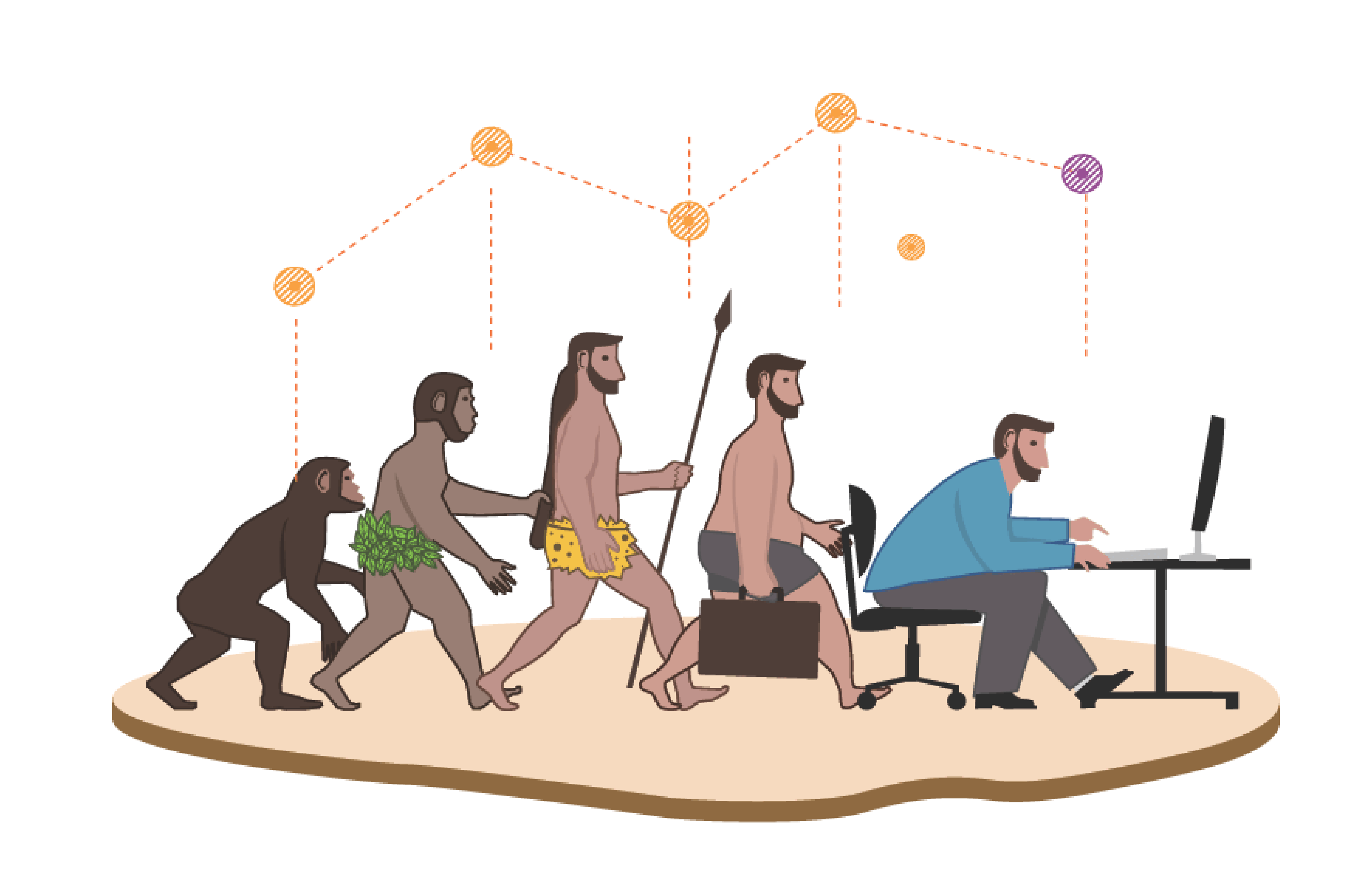


基本理论+经典案例详解深度学习在图像处理中的应用

深度学习技术

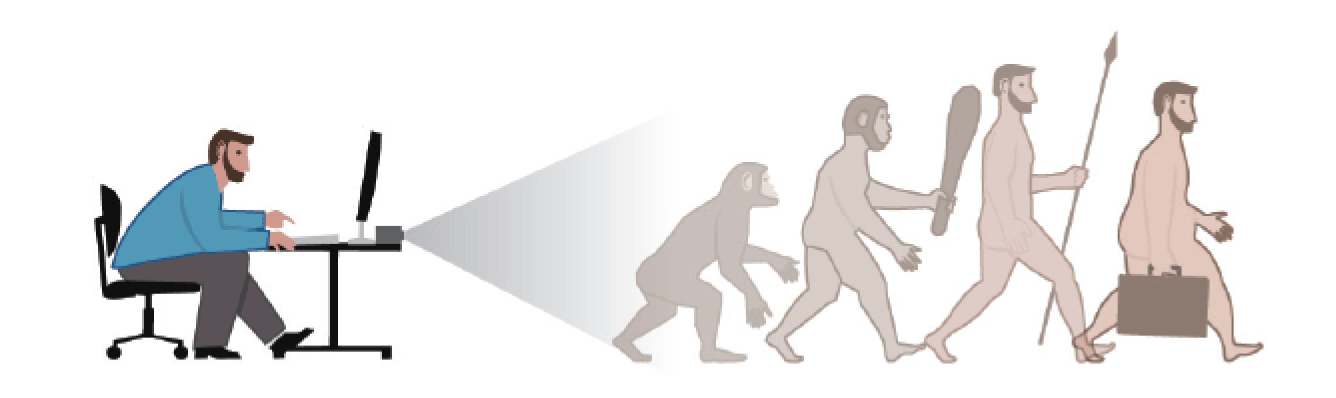
图像处理入门

• 杨培文 胡博强 著 •



理论部分通过具体的简易程序代码讲解取代大量的公式推导
实际应用部分大量讲解参数调整、模型优化的技巧
所有代码均可在云GPU服务器基于容器技术直接执行使用

清华大学出版社



· 杨培文 胡博强 著 ·

深度学习技术
图像处理入门

清華大學出版社
北京

内 容 简 介

本书从机器学习、图像处理的基本概念入手，逐步阐述深度学习图像处理技术的基本原理以及简单的实现。继而以几个实战案例来介绍如何使用深度学习方法，在数据分析竞赛中取得较高的排名。最后，通过一个实战案例，介绍如何将模型放入 iOS 程序，制作相应的人工智能手机App。

本书适用于对深度学习有兴趣、希望入门这一领域的理工科大学生、研究生，以及希望了解该领域基本原理的软件开发人员。此外，本书所有案例均提供了云环境上的代码，便于读者复现结果，并进行深入学习。

图书在版编目（CIP）数据

深度学习技术图像处理入门/杨培文,胡博强著. —北京：清华大学出版社，2018
ISBN 978-7-302-51102-1

Ⅰ. ①深… Ⅱ. ①杨… ②胡… Ⅲ. ①图像处理软件 Ⅳ. ①TP391.413

中国版本图书馆CIP数据核字（2018）第195622号

责任编辑：王金柱
封面设计：王 翔
责任校对：闫秀华
责任印制：宋 林

出版发行：清华大学出版社
网 址：http://www.tup.com.cn，http://www.wqbook.com
地 址：北京清华大学学研大厦A座 **邮 编：**100084
社 总 机：010-62770175 **邮 购：**010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 装 者：北京天颖印刷有限公司
经 销：全国新华书店
开 本：190mm×260mm **印 张：**16.75 **字 数：**422千字
版 次：2018年10月第1版 **印 次：**2018年10月第1次印刷
定 价：69.00元

产品编号：077101-01

序

深度学习是人工智能领域一种新兴的关键技术，其概念由多伦多大学 Hinton 等人于 2006 年提出，含多隐含层的人工神经网络感知器就是一种深度学习结构。深度学习具有很强的学科交叉性，涉及认知科学、脑科学、生物学、数学、运筹学、控制、计算机、通信、大数据、云计算、语言学乃至人文科学、社会科学等学科。近年来，深度学习无论是在基础研究还是工程应用方面都完成了很多创新性工作，例如有监督卷积神经网络、无监督深度置信网络、第一个战胜人类围棋世界冠军的人工智能程序 AlphaGo、人脸识别、语音识别、自然语言处理等。GPU、人工智能芯片等硬件技术的突破，让人们对深度学习的实用化增添了信心。作为全球性的前沿研究问题，深度学习已经受到众多科学家、工程师的关注，其应用几乎涉及工程、金融、法律、文化、娱乐、艺术等社会生活的各个方面，多篇里程碑意义的论文在 Nature、Science 等国际顶尖学术刊物上横空出世，显示出深度学习技术旺盛的生命力和广阔的应用前景。

呈现在读者面前的《深度学习技术图像处理入门》一书，在兼顾介绍深度学习基础理论的同时，更侧重于记录作者本人在学习深度学习技术时的历程、感悟和经验，包括难免走过的弯路，作者也坦承地娓娓道来。通读全书，与同类书籍比较，我认为该书具有“亦师亦友、寓教于乐”的特点：入门部分介绍得通俗易懂，文风严谨亦不失活泼，让高深的理论不那么神秘、枯燥，使得读者更有阅读的乐趣和自信，易于沉浸其中，非常适合作为自学的教材和工具书；案例部分的代码，均基于实际的竞赛，偏重于应用，使得读者也更有兴趣参与这样的竞赛，以“小试牛刀”。

《深度学习技术图像处理入门》是一本能够帮助读者很快掌握和应用深度学习技术进行图像处理的工具书，对生物、医学、安防、机器人等众多图像处理领域的工程应用问题具有很大的参考价值。在我国高度重视人工智能技术的发展、将其上升为国家战略的时代背景下，相信该书的出版，对于推动我国深度学习技术的培育、推广、应用具有积极的作用。

中国农业大学博士生导师 陈建

2018 年 8 月 1 日

前　言

回想 2017 年 4 月，当清华大学出版社的编辑找到杨培文和我，商量着写一本与深度学习相关的书时，我还是比较缺乏信心的。首先，自己本专业是基因组学，或者说是生物学，机器学习方面的知识都是自学的。其次，我根本就没有写过书，由我参与撰写，可能是班门弄斧，内容有误都是小事，万一写的内容给读者灌输了错误的观念、在大方向上误导了初学阶段的读者，实在是难辞其咎。

出版社方面同样了解我们的情况，跟我们说出版社这次想出一本面向非数学、计算机相关专业的书，希望语言更加通俗易懂，例子更贴近实际项目，让非专业出身的人看了以后，对机器学习、图像处理以及深度学习三者有一个最基本的认识。这里，我经常向生物、医学专业背景的人解释机器学习模型的原理，而培文则有多次数据分析竞赛名列前茅的经历，因此出版社希望我们两位尝试一下。

所以接下来编写书籍的过程中，我们的定位就是相比现在市面上主流的相关书籍，前几章写得更加通俗，把入门的门槛再降低一些；然后后面的章节基于参加数据分析竞赛的实战过程，把最终的目标再定高一些；最后我们的配套代码以及环境（http://github.com/Jinglue/DL4Img）要让初学者可以很容易地跑起来，把书籍的内容落在实际运用中。

我们希望这本书可以让非科班出身的读者快速了解深度学习的基本原理，将相关技术举一反三，运用在自己的课题、项目中。以我自己为例，在书籍编写完成后的审阅过程中，我仔细阅读了培文撰写的运用循环神经网络进行验证码识别这一章节（第 10 章）的内容，后来参加百度 AI 挑战赛时，最初的模型就是培文整理的配套代码，后来经过调整，最后取得了第二名的成绩。

最后一点，阅读本书，需要读者具有基本的 Python 编程基础，以及科学计算相关模块的了解。这部分内容本书并未涉及，但读者可以通过斯坦福大学 cs228 相关配套入门习题进行简单的了解，我们对此进行了汉化（https://jizhi.ai/blog/post/cs228-py）。

在此感谢景略集智的王文凯、柯希阳在书籍编写过程中提供的帮助。

胡博强

2018 年 7 月 18 日

目　录

第 1 章

搭建指定的开发环境

“工欲善其事，必先利其器”。在介绍本书的具体内容之前，我们首先需要进行硬件方面的准备，同时安装必要的软件，进而基于这个开发环境学习本书中的案例代码。

本书所用代码及本章内容对应的开发环境，读者可以在 https://github.com/Jinglue/DL4Img 具体查阅。

1.1 为什么要使用指定的开发环境

使用指定开发环境的主要目的是方便读者运行代码。深度学习环境主要基于 Linux 操作系统搭建，此过程需要有 Linux 相关的知识作为铺垫。然而，使用指定的开发环境，可以将这一部分大大简化，使读者直接运行代码，降低入门门槛，否则读者可能会遇到很多无从下手的麻烦，影响主要内容的学习。

首先，确定开发环境的版本。Python 的 Scikit-learn、Tensorflow、Keras 等机器学习、深度学习常用库，很多基本用法在不同版本之间都是不同的。另外，可能一两年后，相比此时 Tensorflow 的语法又有很大不同，到那时读者想使用本书代码的话，就会发现代码运行错误，造成很多困扰。还有这些库的安装过程中，有时并不是安装一个库的问题，同时涉及与系统库

的交互。笔者依稀记得很多年前第一次接触 Python，在大型机的个人用户安装 Scipy 库时，需要通过逐一手动安装系统库来解决包依赖问题，折腾了整整三天。

其次，本书的代码可以在该开发环境下运行。如果读者觉得本书的案例和自己学习工作中的案例比较接近，可以直接使用自己的数据代替书中的数据，或者使用本书的模型快速得到一个和自己工作相关的模型。

最后，本书基础部分（第 2~7 章）的内容，计算量并不大，可以在普通的个人电脑上部署安装，甚至可以打开电脑浏览器，然后直接在景略集智官网里单击与本书内容相关的博客文章，在云端运行程序。当然，本书的案例部分（第 8~11 章）的内容都需要相当的计算量，这就需要准备一台带 GPU 的电脑了，而这样的电脑通常并不便宜。为了方便读者学习，我们专门配置了和本书配套的开发环境，方便读者租用云服务器上手学习。有关电脑配置和云服务器租用方式，详情请见 1.2 节的内容。

1.2 硬件准备

本书主要讨论使用深度学习技术进行图像处理分析，而深度学习技术在模型的训练阶段，需要使用除了 CPU 以外的硬件进行加速才能达到理想的训练速度，实现模型的快速收敛。这些硬件包括 TPU、FPGA、Intel Xeon Phi 处理器等，但目前阶段使用最广泛、开发最简单的还是 GPU，即英伟达（NVIDIA）公司的显卡。

之所以是 NVIDIA 公司的显卡，是因为现在主流深度学习框架大多基于 NVIDIA 的 CUDA 计算库，而 CUDA 计算库支持的硬件主要是自家产品。支持 CUDA 的硬件可以在 NVIDIA 官网中找到（https://developer.nvidia.com/cuda-gpus）。因此，想用自己电脑进行本书深度学习入门学习的用户，如果想借助 GPU 作为硬件帮助，基本要求是需要有一台近几年出品的 NVIDIA 显卡，其显存大小最好在 8GB 以上。

如果用户的电脑没有独立显卡或者显存大小不足，将无法训练本书最后几章中的案例。另外，电脑配置的是 AMD 显卡，是不是必须新买一台电脑呢？重新购置电脑当然也可以，但是毕竟成本比较高，用户同样可以考虑按小时收费，租用 GPU 云服务器。为了方便读者学习，我们为本书的内容在亚马逊云（aws）以及腾讯云（qcloud）上设计了专门的镜像，方便读者使用。

1.2.1 在亚马逊租用云 GPU 服务器

使用 aws us-east-1 节点，GPU 价格是 0.9 美元 / 小时，也可以使用实时竞价（Spot Instance）方式以获得更低的价格。使用方法如下：

（1）登录 aws 官网。进入 https://console.aws.amazon.com/ec2/v2/home?region=us-east-1，单击 Launch Instance，如图 1-1 所示。当然，也可以单击左侧的 Spot Instance 选项选择更低价的实时竞价机器，然后启动。

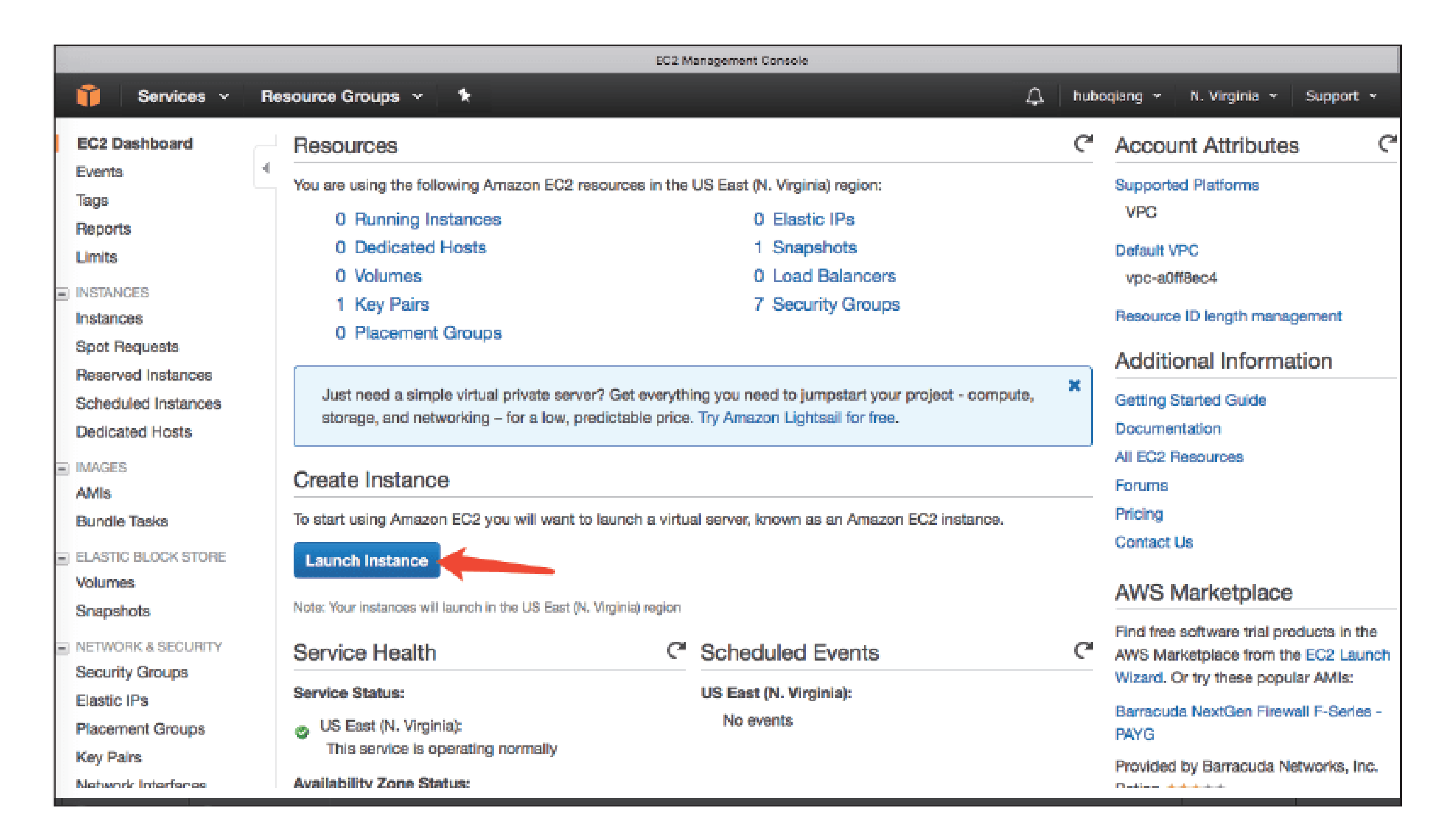


图 1-1　在 aws 上单击 Launch Instance

（2）选择左侧的 Community AMIs 选项，然后在右侧的文本框中查找 NVIDIA Docker，再选择一个镜像，如图 1-2 所示。推荐使用 aws us-east-1 节点。

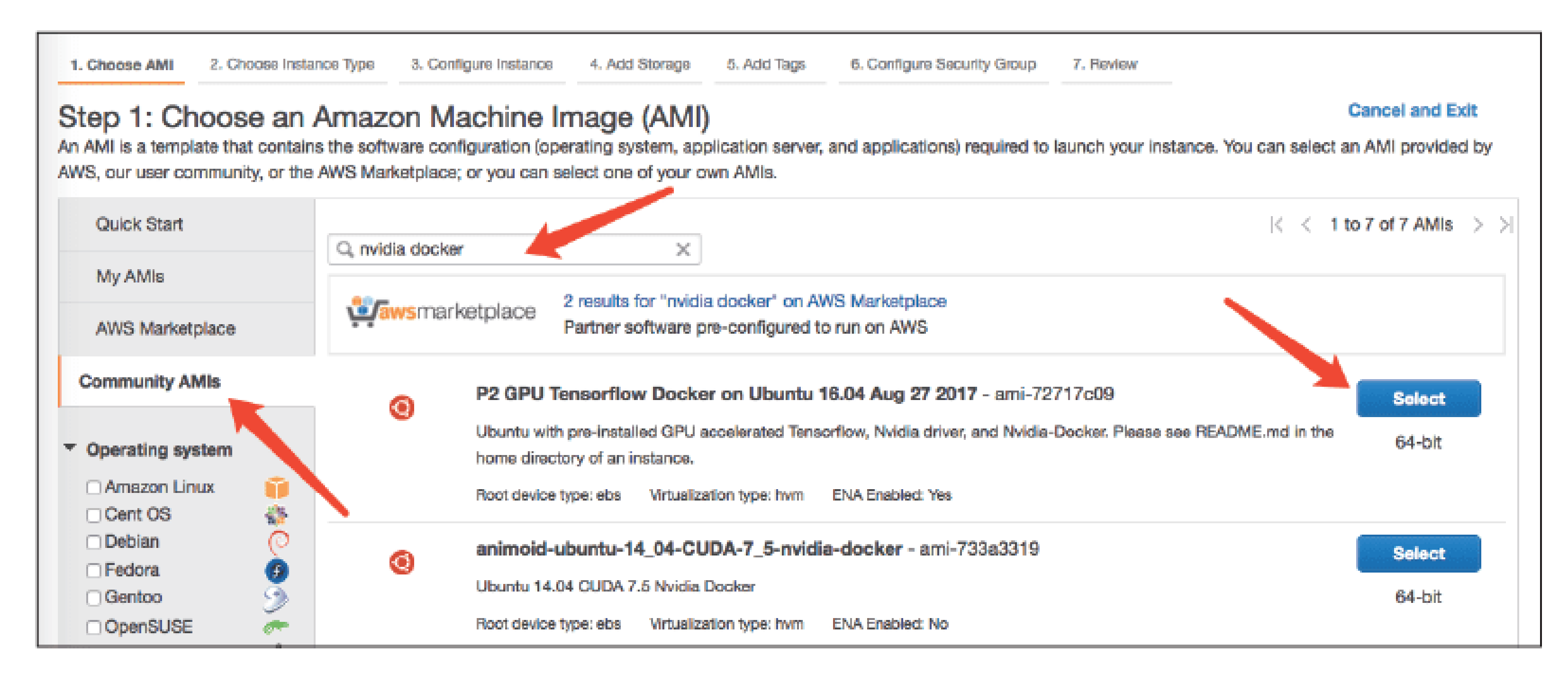


图 1-2　查找预装 NVIDIA Docker 的镜像

（3）选择 p2.xlarge 节点，单击 Review and Launch 按钮启动服务，如图 1-3 所示。

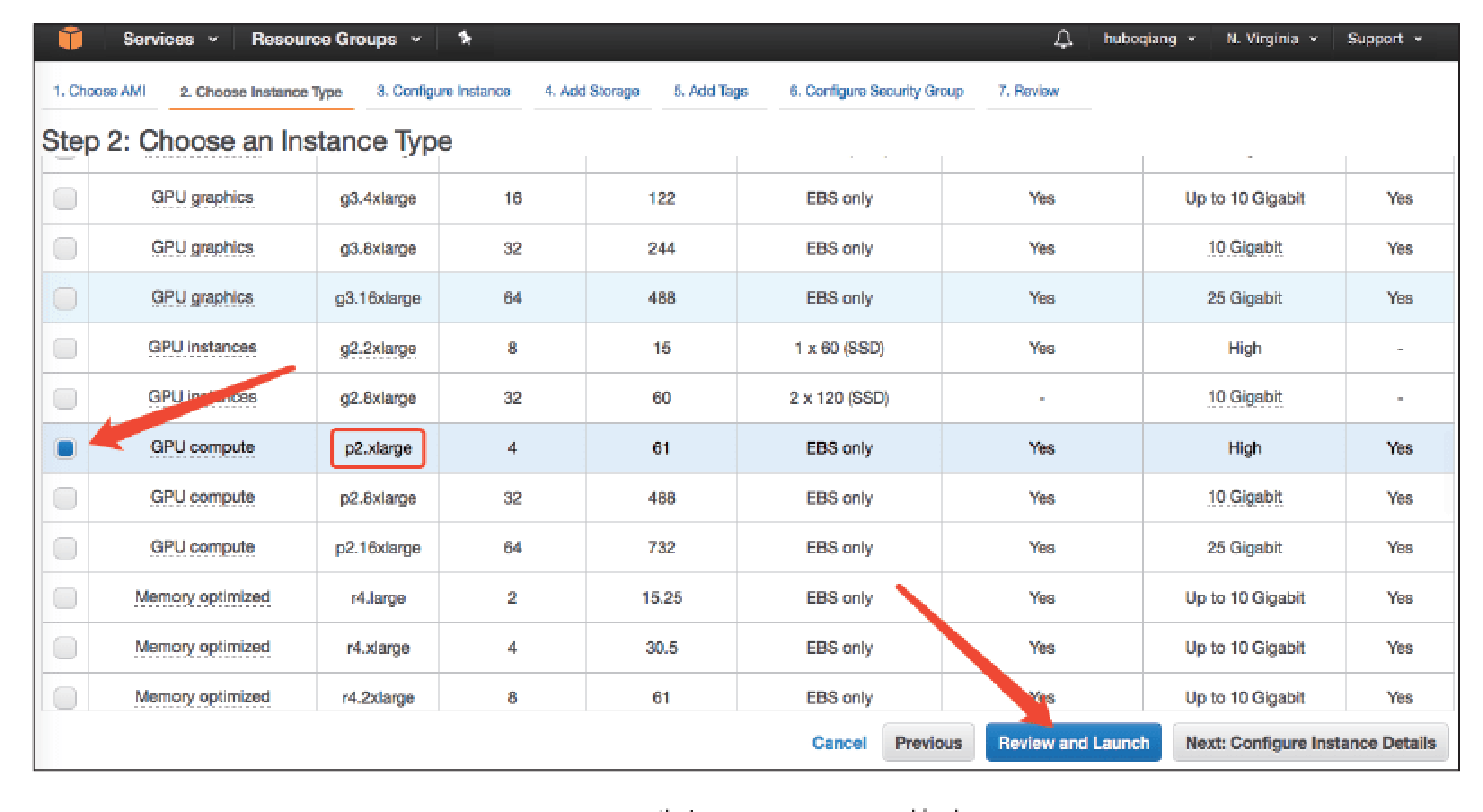


图 1-3　选择 p2.xlarge 节点

（4）接下来选择公钥，如图 1-4 所示。如果还没有公钥就进行创建。创建公钥的过程中注意预留 22、8888、6006 这三个端口，供后续使用。短期使用时，可以选择开启所有端口。

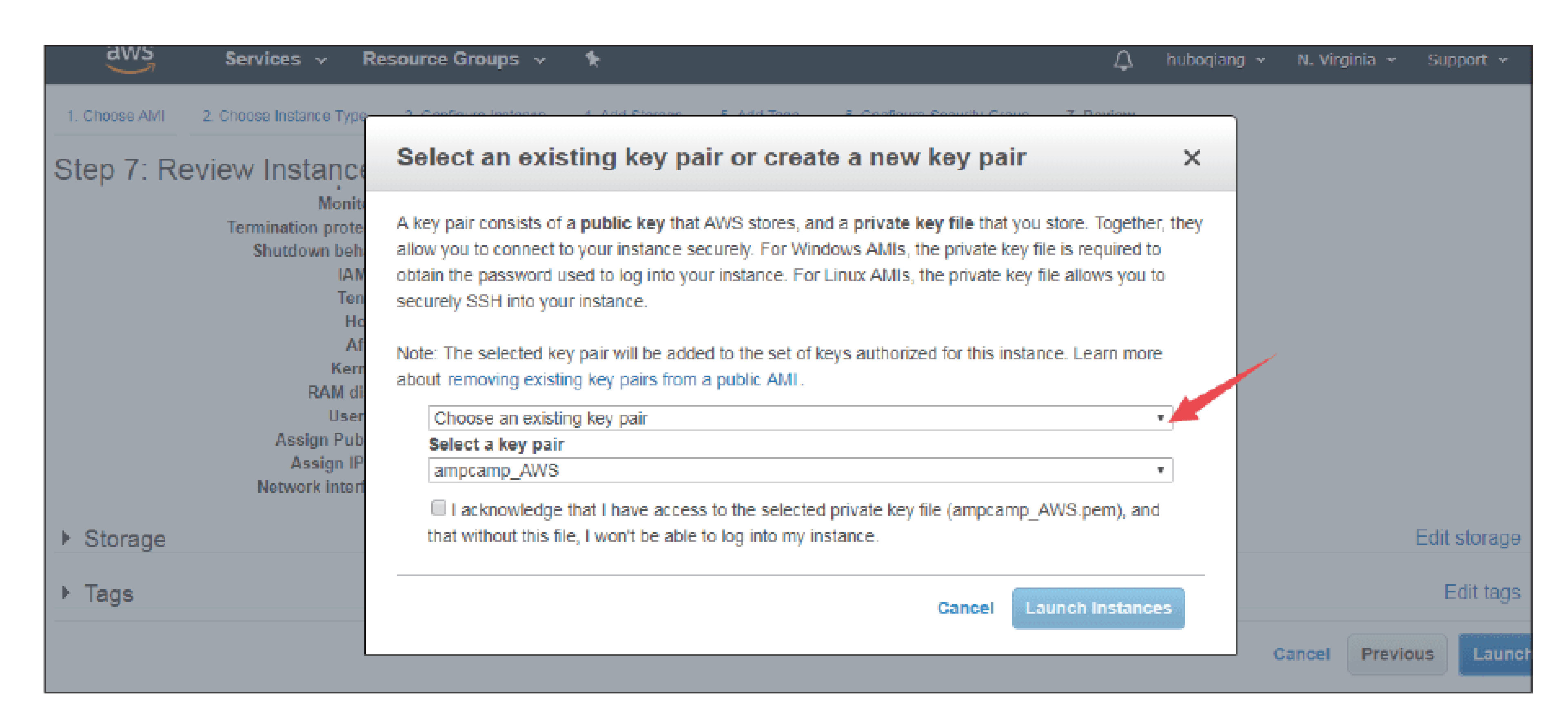


图 1-4 选择公钥

（5）在 PuTTY（Windows）或者终端（Mac/Linux）中，使用密钥启动。这里我们以 PuTTY 为例，需要填写公网地址，如图 1-5 所示。然后需要指定刚才登录时使用的公钥，如图 1-6 所示。

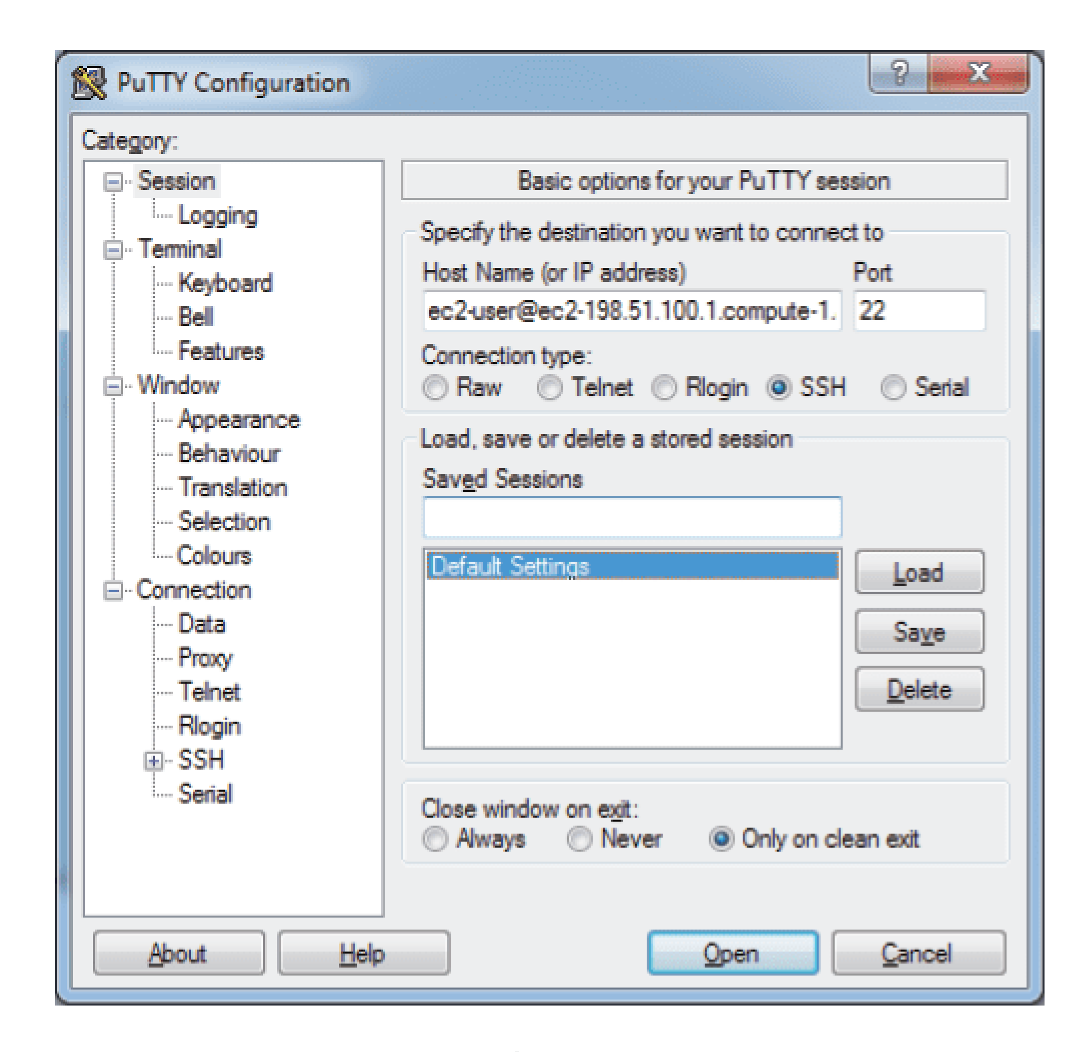


图 1-5 输入公网地址

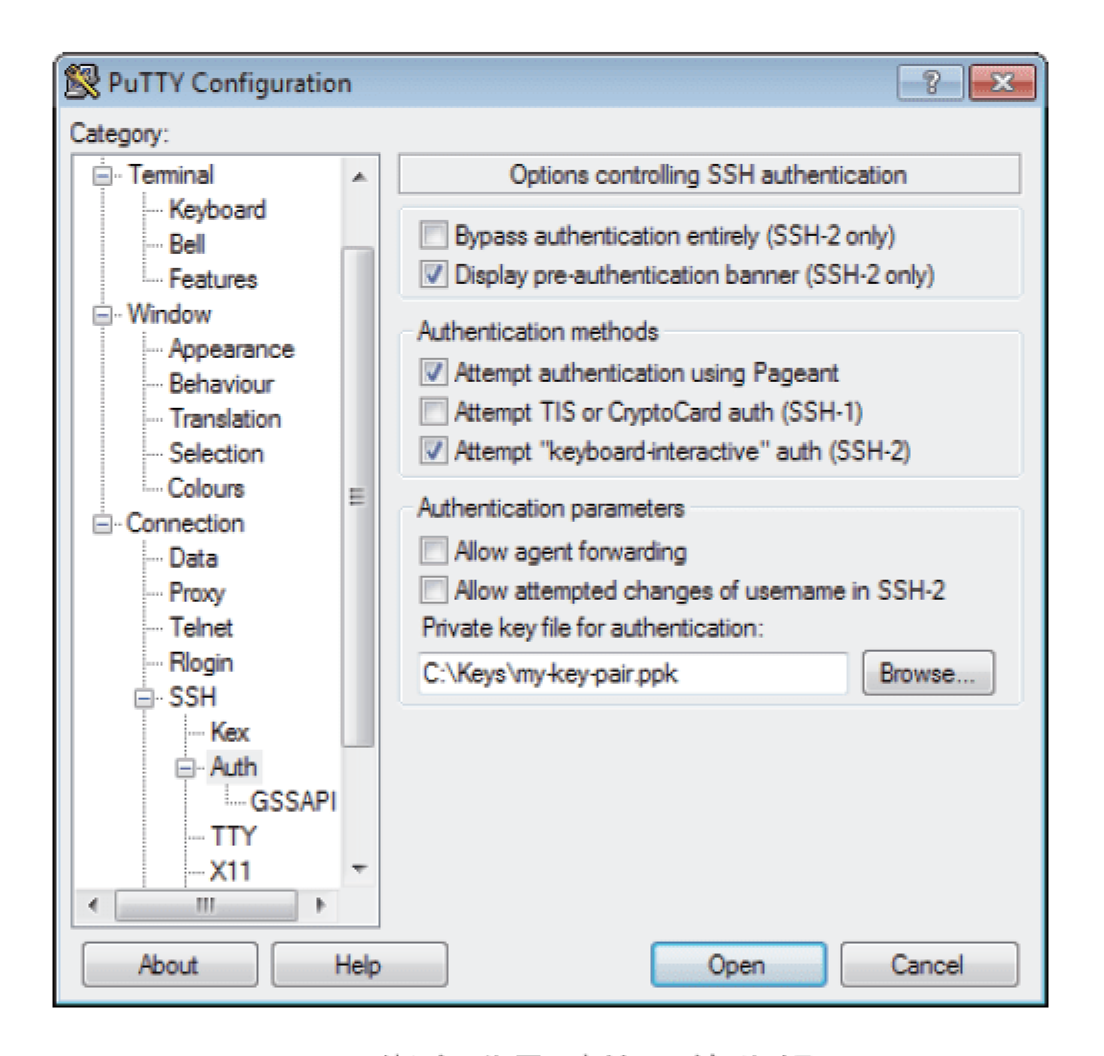


图 1-6 指定登录时使用的公钥

（6）此时单击 Open 按钮，即可登录 aws 服务器。如果遇到登录问题，可以在 aws 上问答（http://docs.aws.amazon.com/zh_cn/AWSEC2/latest/UserGuide/putty.html）中查询具体原因。

1.2.2 在腾讯云租用 GPU 服务器

亚马逊云是国外的机器，大家实际使用时可能网络连接速度并不快。这种情况下，大家可以考虑腾讯云、阿里云的按小时计费。这里以租用腾讯云为例。

（1）进入网址 https://cloud.tencent.com/product/gpu，单击“立即选购”按钮，如图 1-7 所示。



图 1-7　选购 GPU 云服务器

（2）选择“按量计费”，然后指定机型，这里 GPU 单卡足够，如图 1-8 所示。

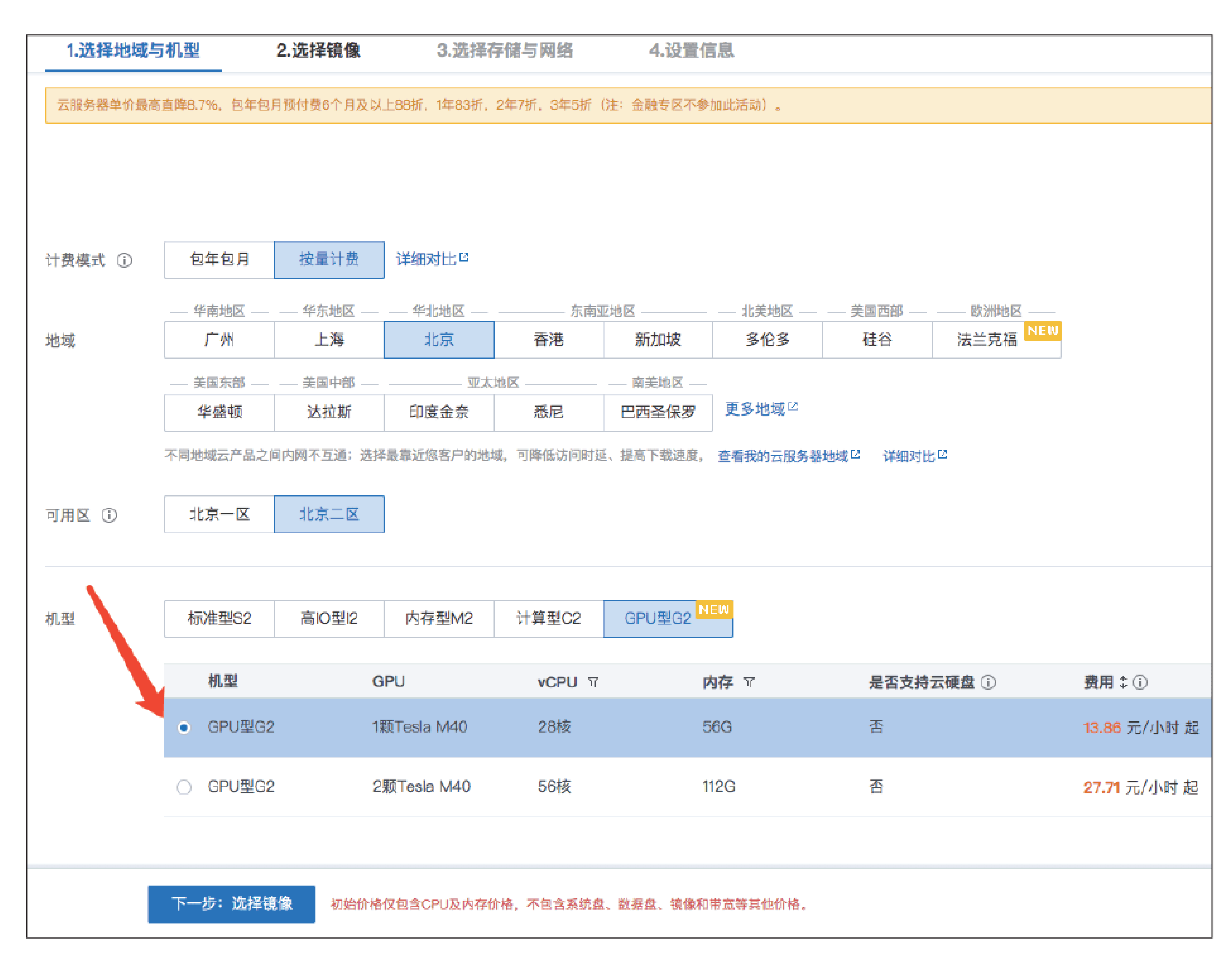


图 1-8　选择计费模式和机型

（3）接下来选择 GPU 镜像，单击“从服务市场选择”链接，如图 1-9 所示。

（4）在弹出的窗口中选择腾讯云官方的 GPU 镜像，如图 1-10 所示。注意，这里显示的官方镜像 CUDA 是 7.5 版本，读者使用该镜像时需要下载并安装 8.0 版本以上的 CUDA。



图 1-9 选择镜像



图 1-10 选择腾讯云官方镜像

（5）选择完毕，单击“下一步：选择存储与网络”按钮，如图 1-11 所示。



图 1-11 单击“下一步：选择存储与网络”

（6）接下来选择带宽和服务器数量等，如图 1-12 所示。

1.选择地域与机型　2.选择镜像　3.选择存储与网络　4.设置信息

系统盘　本地SSD硬盘（本地SSD硬盘固定为50GB，使用本地SSD硬盘的服务器暂不支持升级CPU/内存/硬盘）　选购指引

数据盘　本地SSD硬盘1650G

网络类型　基础网络　私有网络

带宽计费模式　按带宽计费　按使用流量　详细对比

带宽　0Mbps　5Mbps　20Mbps　100Mbps　10 Mbps

分配免费公网IP

服务器数量　1 台

费用：配置费用 14.02 元/小时（阶梯计费 计费详情）　网络费用 1.56 元/小时

上一步　下一步：设置信息

图 1-12 选择带宽和服务器数量等

（7）单击“下一步：设置信息”按钮，设置安全规则、购买服务器。在“安全组”框中至少要保留 22、8888、6006 三个端口，短期使用也可以全部开启，如图 1-13 所示。



图 1-13 设置安全规则和购买服务器

（8）购买成功会显示如图 1-14 所示的页面。稍等几分钟，公网 IP 地址出现后，使用公网 IP 地址登录即可。



图 1-14 购买成功

（9）使用 PuTTY 登录购买的云服务器，其操作步骤与 aws 登录相同，注意不需要设置秘钥，填写 IP 地址即可。

（10）参考 1.3.2 小节，安装 CUDA、Docker、NVIDIA Docker 等。

1.2.3 在云服务器中开启搭载开发环境的 Docker 服务

登录服务器，开启 Docker 以及 NVIDIA Docker 服务，并开启镜像。

```
systemctl start docker
systemctl start nvidia-docker
git clone https://github.com/Jinglue/DL4Img
nvidia-docker pull hubq/dl4img
nvidia-docker run -d -v ~/dl4img/notebook/:/srv -p 8888:8888 -p 6006:6006 hubq/dl4img
```

打开镜像后，读者可以在浏览器中输入下面的内容来访问刚才搭建的开发环境。

```
http://[ 购买云服务器的 IP 地址 ]:8888
```

登录密码为 jizhitencent，如图 1-15 所示。



图 1-15 访问刚搭建的开发环境

1.3 软件准备

如果读者选择购买云服务器，这部分内容就可以省略了，因为云服务器中已经将所需软件安装完成。如果读者自己有符合硬件要求的电脑，建议使用 Linux 的 Ubuntu 16.04 操作系统进行学习。这里以新安装的 Ubuntu16.04 系统以及 CentOS 7 系统为例，介绍如何配置环境。

首先分别介绍如何安装 NVIDIA 显卡驱动程序、Docker 以及 NVIDIA-Docker，我们需要使用这些工具来建立开发环境。

1.3.1 在 Ubuntu 16.04 下配置环境

（1）首先需要安装 NVIDIA 显卡驱动程序。登录 NVIDIA 网站下载驱动程序或打开链接 http://www.nvidia.com/Download/Find.aspx 选择操作系统和安装包。以 M40 为例，选择要下载的驱动程序版本，如图 1-16 所示。

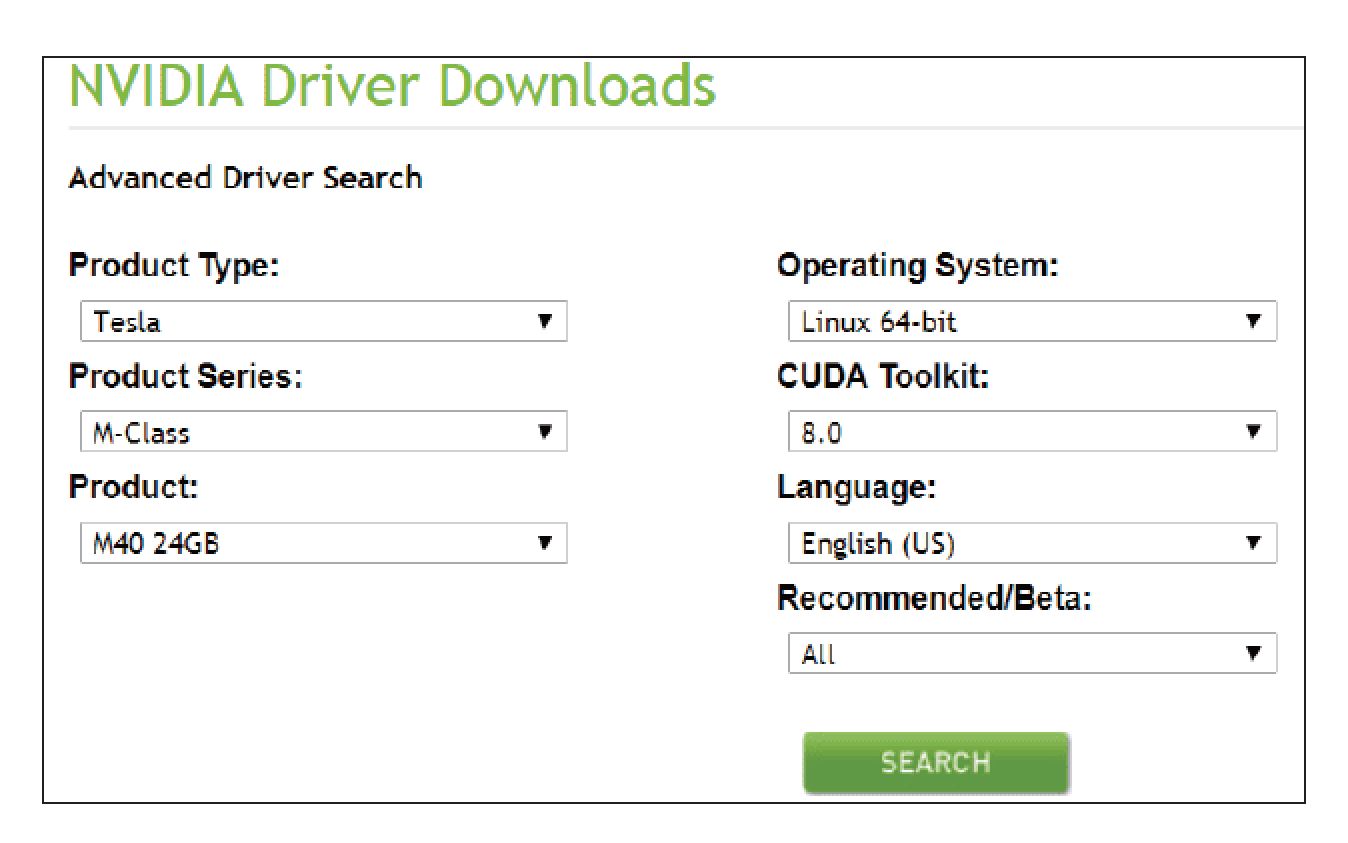
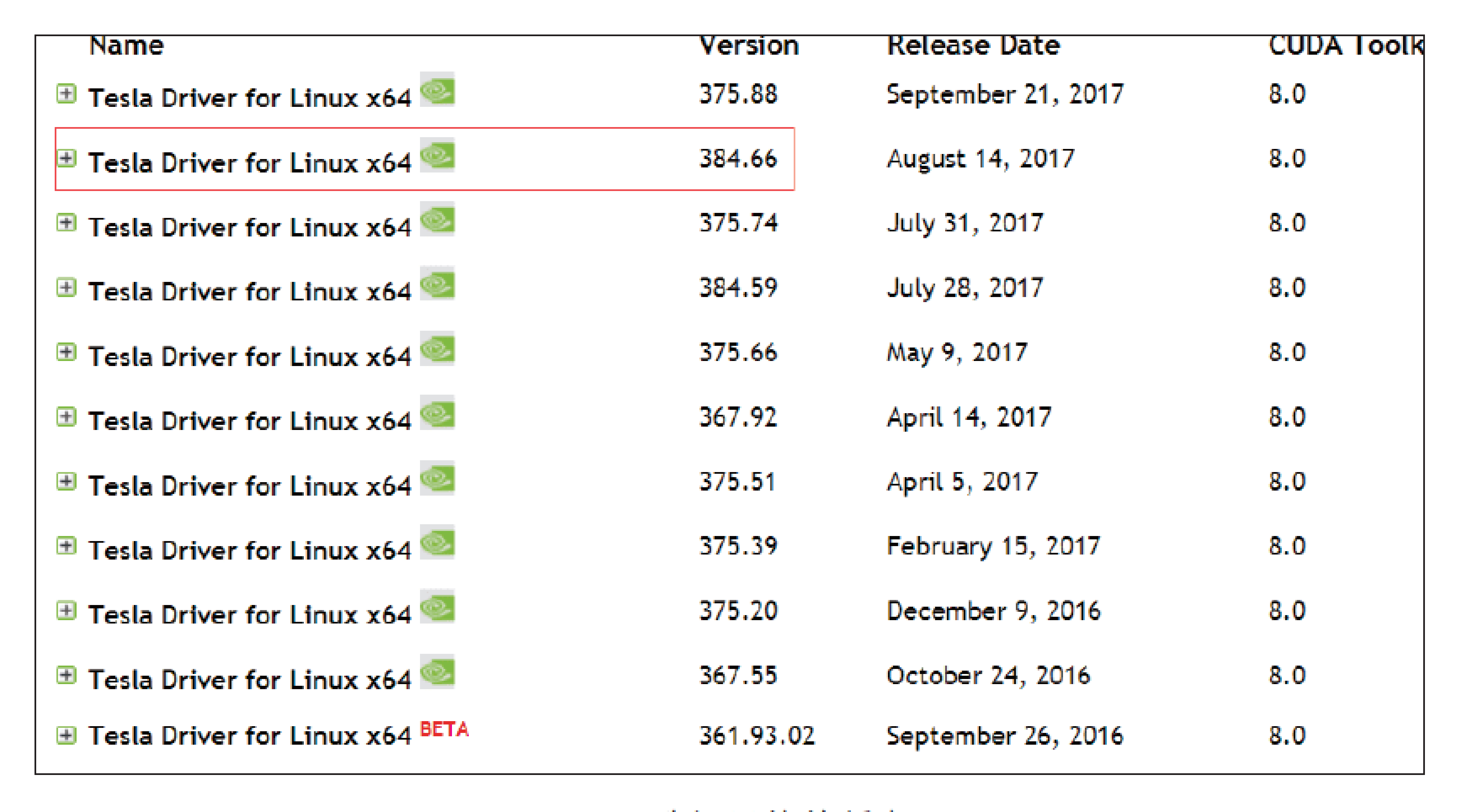


图 1-16　选择 NVIDIA 驱动程序

（2）选择具体的版本号，如图 1-17 所示。



Name	Version	Release Date	CUDA Toolk
Tesla Driver for Linux x64	375.88	September 21, 2017	8.0
Tesla Driver for Linux x64	384.66	August 14, 2017	8.0
Tesla Driver for Linux x64	375.74	July 31, 2017	8.0
Tesla Driver for Linux x64	384.59	July 28, 2017	8.0
Tesla Driver for Linux x64	375.66	May 9, 2017	8.0
Tesla Driver for Linux x64	367.92	April 14, 2017	8.0
Tesla Driver for Linux x64	375.51	April 5, 2017	8.0
Tesla Driver for Linux x64	375.39	February 15, 2017	8.0
Tesla Driver for Linux x64	375.20	December 9, 2016	8.0
Tesla Driver for Linux x64	367.55	October 24, 2016	8.0
Tesla Driver for Linux x64 BETA	361.93.02	September 26, 2016	8.0

图 1-17　选择具体的版本号

（3）在 DOWNLOAD 按钮上单击鼠标右键，在弹出的快捷菜单中选择“复制链接地址”命令，如图 1-18 所示。

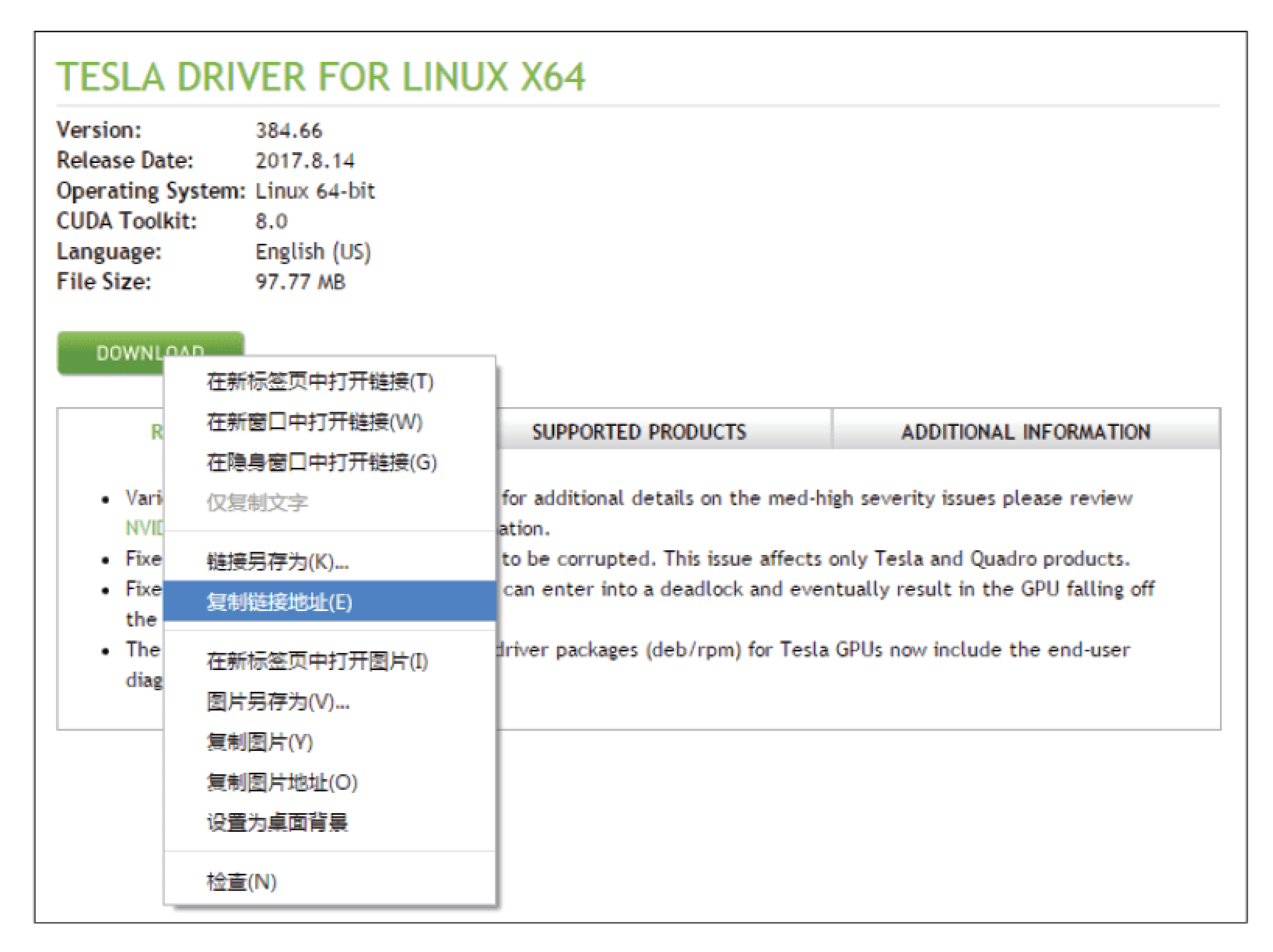


图 1-18 选择“复制链接地址”命令

（4）接着登录 GPU 实例。使用 wget 命令，粘贴上述步骤复制的链接地址下载安装包，如图 1-19 所示；或在本地系统下载 NVIDIA 安装包，上传到 GPU 实例的服务器。

```
[[redacted] ~]# wget http://www.nvidia.com/content/DriverDownload-March2009/confirmation.php?url=/XFree86/Linux-x86_64/384.66/NVIDIA-Linux-x86_64-384.66.run&lang=us&type=Tesla
```

图 1-19 使用 wget 命令

（5）对安装包添加运行权限，例如对文件 NVIDIA-Linux-x86_64-384.66.run 添加运行权限：

```
chmod +x NVIDIA-Linux-x86_64-384.66.run
```

（6）安装当前系统对应的 gcc 和 kernel-devel 包：

```
sudo yum install -y gcc kernel-devel-xxx
```

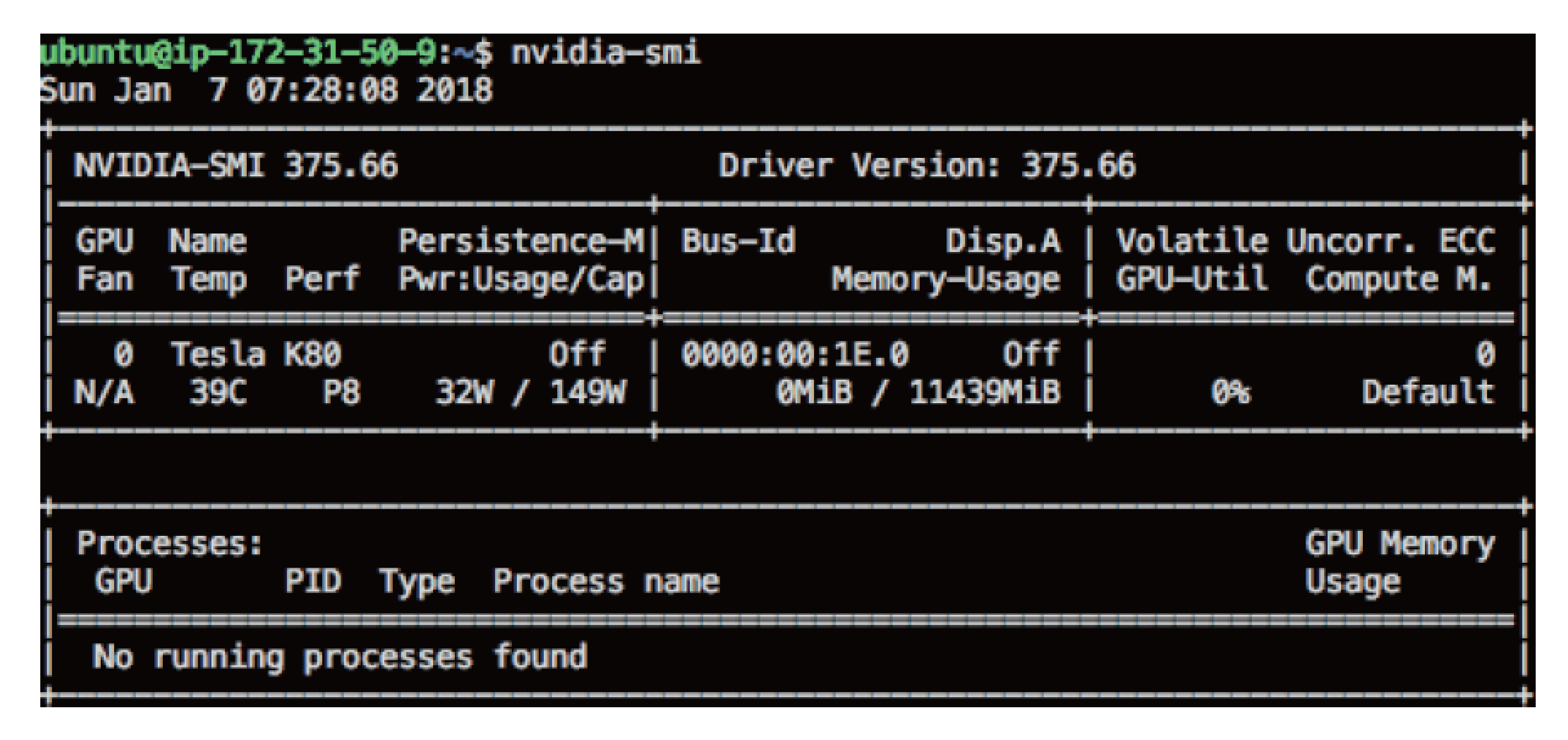
xxx 是内核版本号，可以通过 uname -r 查看。

（7）运行驱动安装程序：

```
sudo /bin/bash ./NVIDIA-Linux-x86_64-384.66.run  其他参数
```

按照提示进行后续操作。

（8）安装完成后，在终端输入 nvidia-smi，如果有类似如图 1-20 所示的 GPU 信息显示出来，说明驱动程序安装成功。


```
ubuntu@ip-172-31-50-9:~$ nvidia-smi
Sun Jan  7 07:28:08 2018
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 375.66                 Driver Version: 375.66                    |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla K80           Off  | 0000:00:1E.0     Off |                    0 |
| N/A   39C    P8    32W / 149W |      0MiB / 11439MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID  Type  Process name                               Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

图 1-20　显卡驱动程序安装成功

（9）接下来，安装 CUDA、Docker 以及 NVIDIA-Docker，并获取本书镜像：

```
    # 如果未安装 CUDA 或者 CUDA 版本低于 7.5，需要安装 CUDA
    wget https://developer.nvidia.com/compute/cuda/8.0/Prod2/ local_
installers/cuda-repo-ubuntu1604-8-0-local-ga2_8.0.61-1_amd64-deb
    sudo dpkg -i cuda-repo-ubuntu1604-8-0-local-ga2_8.0.61-1_amd64.deb
    sudo apt-get update
    sudo apt-get install cuda

    # 安装 Docker
    sudo apt-get update
    sudo apt-get install apt-transport-https ca-certificates
    sudo apt-key adv --keyserver hkp://p80.pool.sks-keyservers.net:80
--recv-keys 58118E89F3A912897C070ADBF76221572C52609D

    sudo echo "deb https://apt.dockerproject.org/repo ubuntu-xenial main" >
/etc/apt/sources.list.d/docker.list
    sudo apt-get update
    sudo apt-get install docker-engine
    # 安装 NVIDIA-Docker
    wget -P /tmp https://github.com/NVIDIA/nvidia-docker/releases/download/
v1.0.1/nvidia-docker_1.0.1-1_amd64.deb
    sudo dpkg -i /tmp/nvidia-docker*.deb && rm /tmp/nvidia-docker*.deb

    # 启动 Docker 服务
    systemctl start docker
    systemctl start nvidia-docker

    # 下载镜像
    sudo nvidia-docker pull hubq/dl4img
```

1.3.2 在 CentOS 7 下配置环境

根据腾讯云官方文档指导，安装 NVIDIA 显卡驱动。这里的方法与 Ubuntu 16.04 相同。相关文档指导请参考 https://www.qcloud.com/document/product/560/8048。

```
# 安装 CUDA
wget https://developer.nvidia.com/compute/cuda/8.0/Prod2/local_installers/cuda-repo-rhel7-8-0-local-ga2-8.0.61-1.x86_64-rpm
rpm -i cuda-repo-rhel7-8-0-local-ga2-8.0.61-1.x86_64-rpm
yum install cuda

# 安装 Docker
yum install docker

# 安装 NVIDIA-Docker
wget https://github.com/NVIDIA/nvidia/docker/releases/download/v1.0.1/nvidia-docker-1.0.1-1.x86_64.rpm
rpm -i nvidia-docker-1.0.1-1.x86_64.rpm

# 启动 Docker 服务
systemctl start docker
systemctl start nvidia-docker

# 下载镜像
####################################################################
## 如果使用腾讯云 CentOS 7 GPU 服务器，这里建议换为腾讯云 Docker 源。#
## 需要修改 Docker 配置文件 /etc/sysconfig/docker，添加:             #
## OPTIONS='--registry-mirror=https://mirror.ccs.tencentyun.com'    #
####################################################################
sudo nvidia-docker pull hubq/dl4img
```

1.4 参考文献及网页链接

［1］ Amazon Elastic Compute Cloud. Available at: http://docs.aws.amazon.com/zh_cn/AWSEC2/ latest/UserGuide/.

［2］ CUDA GPUs. NVIDIA Developer (2017). Available at: https://developer.nvidia.com/cuda-gpus.

［3］ Nvidia. NVIDIA/nvidia-docker. GitHub (2017). Available at: https://github.com/NVIDIA/nvidia-docker.

第 2 章

温故知新——机器学习基础知识

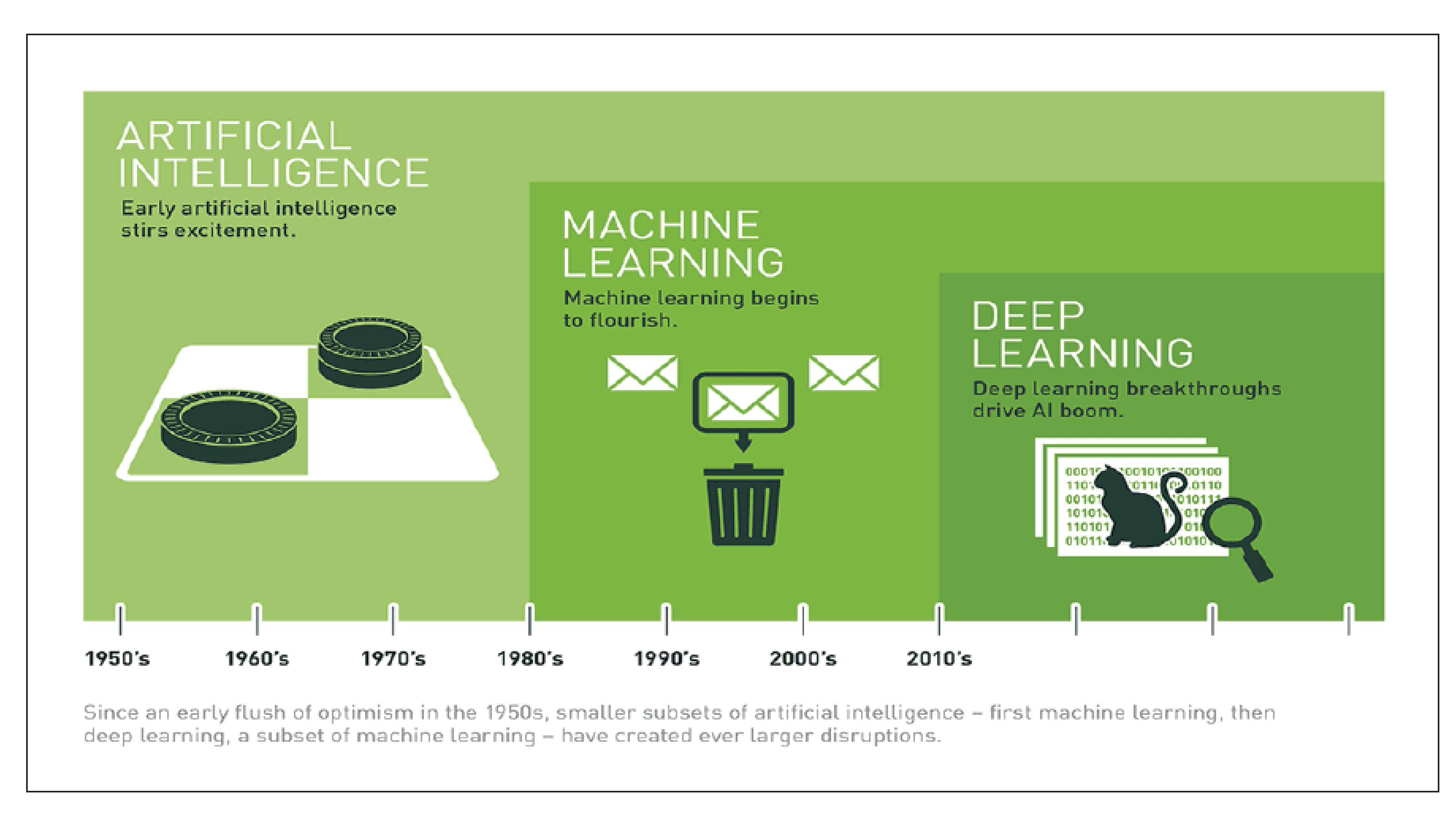
完成开发环境的配置后，我们开始介绍基于深度学习的图像处理技术。在正式讨论本部分内容之前，先给不熟悉深度学习基本概念以及图像处理技术的读者简要介绍一些必备的基础知识。本章主要介绍机器学习的相关概念，其中前几节讨论机器学习与深度学习的共性，后几节讨论深度学习相比传统机器学习方法做了哪些提升。

此外，如果读者对于 Python 基本编程以及科学计算相关知识缺乏了解，阅读本章代码感到理解起来有困难，可以先用斯坦福大学 cs228 课程中的一个快速入门教程（http://cs231n.github.io/ python-numpy-tutorial/），短时间内熟悉相关概念。同时，景略集智网站上也提供了原版教程的中文翻译（https://jizhi.im/blog/post/cs228-py），供读者学习。

2.1 人工智能、机器学习与深度学习

近年来，深度学习的概念十分火热，人工智能也由于这一技术的兴起吸引了越来越多的关注。我们将结合一些基本的用例，简要介绍一下这个新的技术。

首先需要明确人工智能、机器学习以及深度学习三者之间的关系，如图 2-1 所示。如 NVIDIA 官网文章所述，人工智能是一个非常大的概念，而机器学习只是人工智能的一种实现方法。深度学习同样也是一种实现机器学习的方法，是在机器学习的基础上建立起来的。首先，从字面上看，二者都是在“学习”，因此在评价深度学习训练出的模型好坏时，同样直接来源于机器学习的评价方法。其次，深度学习最基本的形式是深度神经网络，直接脱胎于机器学习中的神经网络模型。



（图片来源：https://blogs.nvidia.com/blog/2016/07/29/whats-difference-artificial-intelligence-machine-learning-deep-learning-ai/）

图 2-1 人工智能、机器学习以及深度学习三者之间的关系

正是由于深度学习直接脱胎于机器学习理论，因此本书将首先介绍一些基本的机器学习知识。机器学习本身包含了很多内容，如果对其进行简单的分类梳理，可以归于以下几类：

- 非监督学习
  - 聚类
  - 降维
  - ...
- 监督学习
  - 回归问题
  - 分类问题

单纯讲概念，可能看起来有些枯燥，我们不妨把机器学习和人类的学习行为类比一下。监督学习相比非监督学习最大的区别是，这种方法有**明确的评价指标**，这种指标类似学校里的考试成绩，我们可以简单地认为考试成绩高，这个学生就是好学生。对于机器而言，就是机器训练的模型在给定的数据集中，预测准确率高，这个模型就是一个好模型。因此我们不妨认为监督学习就是一种唯分数论的应试教育方法，参见表 2-1。

表2-1 监督模型

应试教育	监督模型
学生	数学模型
卷子	数据集
考试分数	预测准确率

有应试教育，就有素质教育。素质教育并非没有评价标准，但是相比应试教育要宽松很多，在考察过程中，手里可以有更多的主观因素。这一点在非监督学习中同样成立。如一句古话所言，“近朱者赤，近墨者黑”，要评价一个人如何，就看他平时和什么样的人在一起。非监督学习中的各种**聚类**方法，同样使用了这种思想，就是并不直接评价某一个体，而是看个体之间的接近程度，将众多个体归为少数几个群体，再基于这个群体的特征进行简要概括。

试卷中有主观题和客观题，我们用来类比的监督学习同样可以分为这两种。我们知道主观题的答案，如语文阅读、政治历史问答题，是不要求跟标准答案完全一致的，其评价标准也是越接近越好；这一点就类似机器学习的**回归**问题，比如用模型预测房价、股市走势，大致预测出价格趋势就非常了不起了，不可能圆、角、分全部正确才认为预测正确，差一分就预测错误。而简单的客观题，如判断题、单项选择题，只有固定数目选项，必须和标准答案完全一致才算正确的，这一点类似于机器学习的**分类**问题，比如预测一个人是否患有某种疾病，有就是有，没有就是没有。

注意，这里有一个误区，即认为素质教育优于应试教育。现在机器学习领域中的吴恩达等学者也一再强调非监督算法的重要意义，实际上拿到一个学习任务后，具体使用哪一种方式去分析还是需要考虑应用场景的。通常我们不了解这个学习任务的目的性、需要找线索时，会用非监督找线索，包括聚类、降维方法等。如果明确了学习的目的性，追求高准确率，就需要使用监督学习的方法了。

由于本书是入门读物，并且希望给读者带来快速上手的体验，因此我们将在接下来的过程中主要介绍监督学习的分类部分。

2.2 训练一个传统的机器学习模型

运用机器学习方法分析数据、建立预测模型本身是一个非常复杂的过程，很难在较短的篇幅内完全说清楚，所以这里我们将会结合一个简单的实战案例，将基本概念结合代码实现出来。同时，本节也将穿插介绍如何使用 Python 的数据科学套装，如表 2-2 所示。

表2-2 使用Python数据的科学套装

Package	开发环境的版本	作　用
Python	3.5.2	Python主程序
Jupyter	1.0.0	Python的浏览器端开发环境

（续表）

Package	开发环境的版本	作　用
Numpy	1.12.1	矩阵计算库
Scipy	0.19.0	高级科学计算
Pandas	0.20.1	数据结构（类似SQL）
Sklearn	0.18.1	机器学习
Matplotlib	2.0.1	基础绘图
Seaborn	0.7.1	高级绘图

如果使用了第 1 章中指定的开发环境，这里直接使用就好，否则需要注意一下，版本号的偏差可能会造成程序报告警告，说某一用法是以前的，未来版本会抛弃，也有可能会直接报错。如果本章接下来的程序提示错误，请大家首先确认使用的是否为指定的开发环境以及正确的版本号。

2.2.1 第一步，观察数据

我们这里使用 sklearn 官方提供的鸢尾花分类数据集。这个数据集最初是埃德加·安德森从加拿大加斯帕半岛上的鸢尾属花朵中提取的地理变异数据，包含 150 个样本，属于鸢尾属下的三个亚属，即山鸢尾（setosa）、变色鸢尾（versicolor）和维吉尼亚鸢尾（virginica）。四个特征被用作样本的定量分析，分别是花萼（Sepal）和花瓣（Petal）的长度和宽度。这个案例的目的是通过**建立一种数学模型，尝试使用四个特征去预测某一鸢尾花属于哪一个亚种。**具体如何建立这种模型，接下来我们将会逐步讲解。

要建立模型，首先需要观察数据，而观察数据的第一步是要阅读数据相关的说明文档，弄清楚我们拿到的数据是哪一种具体格式。虽然我们知道了数据有多少个样本、多少种特征，但是目前还不清楚数据是以什么格式给我们的，是文件还是数据库，或者是一个 python 对象，所以需要去 sklearn 官网查阅说明文档，地址是 http://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_iris.html。

文档提到返回的对象如下：

Returns:

```
    data : Bunch
    Dictionary-like object, the interesting attributes are: 'data',
the data to learn, 'target', the classification labels, 'target_
names', the meaning of the labels, 'feature_names', the meaning of
the features, and 'DESCR', the full description of the dataset.
    (data, target) : tuple if return_X_y is True
    New in version 0.18.
```

文档信息写到，这是一个 Python 对象（object），并且具有字典（dictionary）的特征。具体的调用方法如下：

```
from sklearn.datasets import load_iris
data = load_iris()
```

2.2.2 第二步，预览数据

既然上一步提到这是一个具有字典特征的对象，我们就可以用 Python 字典调用的方法对数据有一个总体的预览。

这里首先简单介绍 Python 的字典。Python 主要包括列表、元组以及字典这几种较为高级的数据结构，如果读者熟悉 C++，其实就是 C++ 标准库里的 vector、set 以及 unordered_map。字典结构其实就是一个键值对（unordered_map），但这里的键值对相比 C++ 而言，用起来相对容易一些，可以方便地指向字符串，甚至列表、数组和其他字典：

```
map_abc = {
    "a" : 0,
    "b" : [1,2,3],
    "c" : {"cc" : 4}
}
print(map_abc["a"])
print(map_abc["b"], map_abc["b"][1])
print(map_abc["c"]["cc"])
```

运行结果：

```
#out:
0
[1, 2, 3] 2
4
```

可以使用 for 循环遍历整个字典：

```
for k in map_abc:
    print("key:",k, "\nvalue:", map_abc[k])
```

运行结果：

```
#out:
key: b
value: [1, 2, 3]
key: a
value: 0
key: c
value: {'cc': 4}
```

同样，也来遍历我们的数据：

```
from sklearn import datasets
data = datasets.load_iris()
for k in data:
    print("############\n##%s##\n############\n" % k)
    print(data[k])
```

data.data 就是需要的 150×4 的输入特征矩阵，四个特征存在 data.feature_names 里。分类标签在 data.target 中，其中的 0、1、2 分别代表 data.target_names 里面的 setosa、versicolor 以及 virginica。由此，我们完成了数据的基本预览。

实际上，在预览阶段还有很多问题需要思考：

- 不同分类的样本分类是否均匀？
- 数据是否受到极值、缺失值的影响？
- 能否不建立模型就看出三者的分类关系？
- 能否对样本的分类情况进行简单的预览？

这些问题可以用很简单的程序回答。其实我们想更快地回答这些问题，因此接下来将引入 Python 数据分析套装，用 pandas 对数据进行类似 SQL 的合理结构化，然后基于可视化分析库 matplotlib 与 seaborn，通过图形回答这些问题。

对矩阵格式的数据进行结构化处理，转换为 pandas 的 dataframe。

```
df = pd.DataFrame(data.data)
df.columns = data.feature_names
df['species'] = [ data['target_names'][x] for x in data.target ]
df.head()
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	Species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5	3.6	1.4	0.2	setosa

不同分类的样本分类是否均匀？

对 species 分组计数：

```
df_cnt = df['species'].value_counts().reset_index()
df_cnt
```

结果如下：（注意这里不加 reset_index 的话，将不会返回数据框的形式，不方便制图。）

	index	species
0	setosa	50
1	virginica	50
2	versicolor	50

对结果做图：

```
sns.barplot(data=df_cnt, x='index', y='species')
```

其结果如图 2-2 所示。

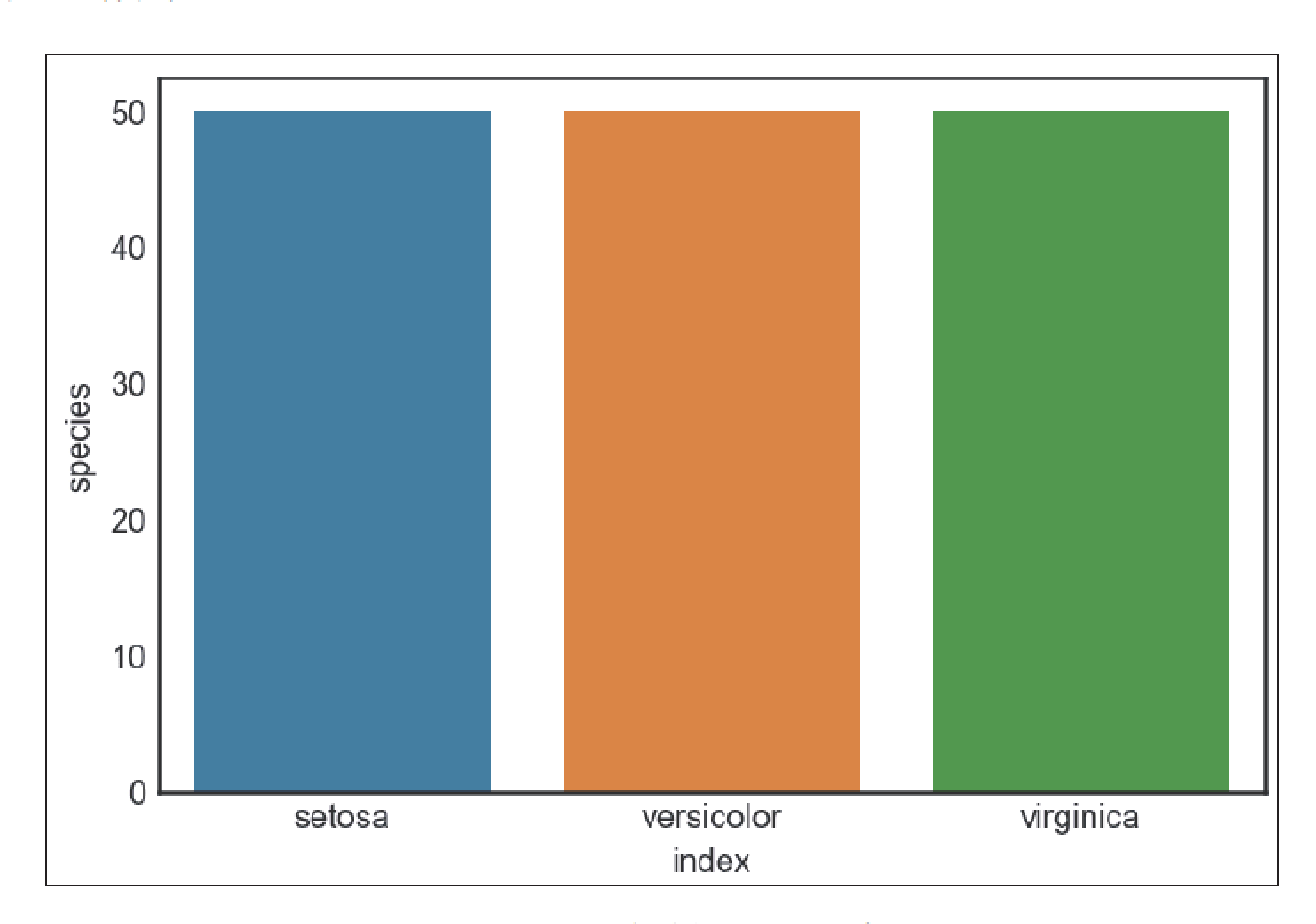


图 2-2　分组计数结果做图效果

数据是否受到极值、缺失值的影响？

极值、缺失值会影响对数据的整体认识，对模型产生干扰，因此，我们在正式分析数据之前，首先应该关注它们会产生什么样的影响。这里的极值是指相比其他数字而言，非常大或者非常小的值，又称为离群值，既可能是由于收集阶段的错误造成的，也可能是由于一些意外因素造成的。极值会对数据分析结果造成一定干扰，典型的例子如同国家统计局发布人均年收入数据时，很多人发现自己“被平均”，因此对于极值，会根据实际需求选择是否剔除。

缺失值同样也会有影响。比如小明给自己量体温，量了一周的体温值：

时间	周一	周二	周三	周四	周五	周六	周日
体温	36.2	36.3	36.4	忘测了	36.3	36.2	36.4

问小明这一周的平均体温是多少？这种情况下，真实的平均体温我们是不知道的，因为不知道小明周四的体温是多少，所以整周的平均值也是不知道的：

```
import numpy as np
np_temp = [36.2, 36.3, 36.4, np.nan, 36.3, 36.2, 36.4]
np.mean(np_temp)
```

运行结果：

```
# out:
  nan
```

遇到这种情况，实际上是对数据的一种浪费，因为周四没有收集到体温数据，这一周其他6天的数据就失去了作用。在大规模的数据中，我们很难保证所有数据都被合理地收集到，此时如果有几万个数据，因为个别的缺失就全部扔掉，这种浪费就更加明显了。对于这种情况，通常的做法是在计算的情况下不考虑未知量，或者是用平均值、中位数去填补未知值。

方法 1 不考虑周四

```
np_temp = [36.2, 36.3, 36.4, np.nan, 36.3, 36.2, 36.4]
np.nanmean(np_temp)
```

运行结果：

```
# out
36.3...
```

方法 2 用其他数字平均值填补缺失值后计算

```
np_temp[3] = np.nanmean(np_temp)
np.mean(np_temp)
```

运行结果：

```
# out
36.3...
```

方法 3 用数字 0 填补缺失值后计算（大错特错，不要这样）

```
np_temp[3] = 0
np.mean(np_temp)
```

之所以把这一条写上来，是因为现实中真的有人会用 0 直接填补缺失值。对这种情况，不是说不行，很多时候其实可以用 0 填补，比如统计某人口稀少地区的各种汽车销量，如果某一汽车销量缺失，我们是可以用 0 填补的，因为这个数字可能真的很小，可能真的是 0，以至于统计人员收集数据时直接忽略了这种汽车。但小明周三的体温，实在不太像是 0 度，因此这里不可以用 0 来填补缺失值。

介绍完极值与缺失值将会带来的问题后，我们想知道如何用最快的速度判断数据里面是否存在这两种情况，最简单的办法就是对数据框执行 .describe() 操作：

```
df.describe()
```

运行结果：

```
# out:
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)
count	150	150	150	150
mean	5.843333	3.054	3.758667	1.198667
std	0.828066	0.433594	1.76442	0.763161
min	4.3	2	1	0.1
25%	5.1	2.8	1.6	0.3
50%	5.8	3	4.35	1.3
75%	6.4	3.3	5.1	1.8
max	7.9	4.4	6.9	2.5

这里的极值（min/max）看起来并不明显，平均值、标准差范围也比较接近，并且不存在数据缺失。如果有缺失，最后一行会单独显示缺失了几个值，继而我们进一步确认这四个特征是否为正态分布。这种检验通常使用 QQ-Plot，即横坐标是标准正态分布的 Quantile（如图 2-3 所示的 x 轴），纵坐标是样本实际分布的 Quantile，如果实际分布是正态分布，两者之间就会存在线性的关系。

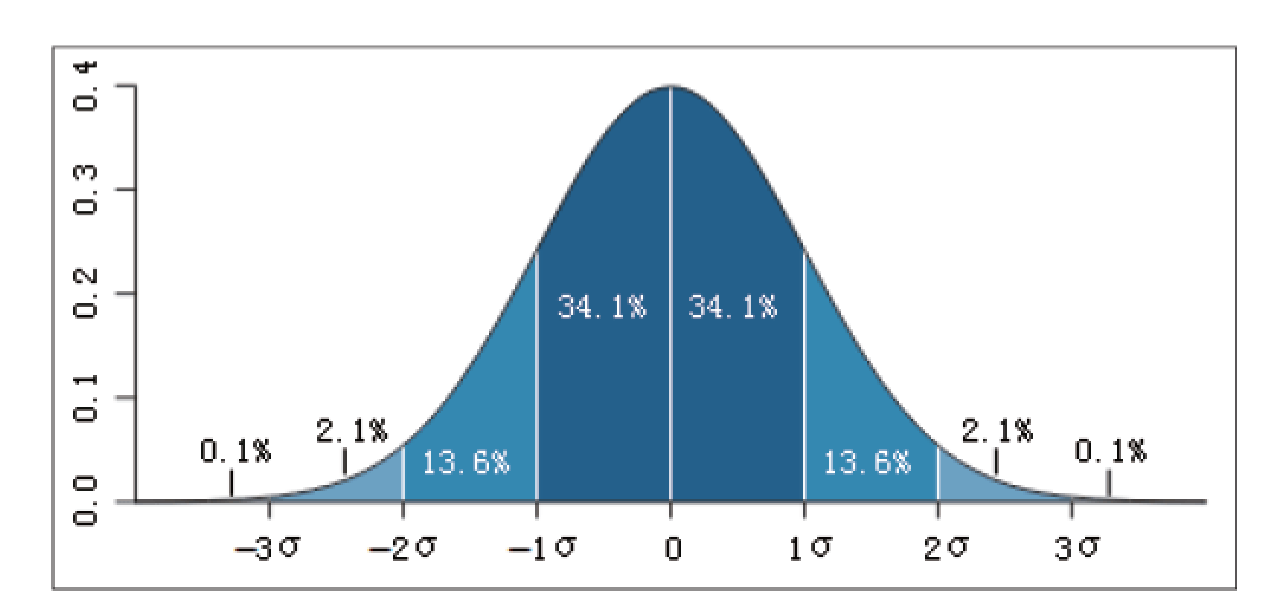


图 2-3　横坐标是标准正态分布的 Quantile

对我们的四个特征进行正态分布检验：

```
from scipy import stats

for i in range(4):
    name = data.feature_names[i]
    ax = plt.subplot(2,2,i+1)
    stats.probplot(df[name], plot=ax)
    ax.set_title(name)
```

运行结果如图 2-4 所示。

这里的前两个特征可以认为来自同一个正态分布，如此很多统计假设（如 t 检验）等就可以成立，继而在后续的建模阶段方便我们使用很多有效的算法。后两个特征更像是两条线中间断开了，似乎是来自两个正态分布。这种情况未必是坏事，如果某个分类种类完全来自其中一个正态分布，其他的来自另一个正态分布，那么这个特征就成了一个非常具有区分度的特征，需要我们重点关注。

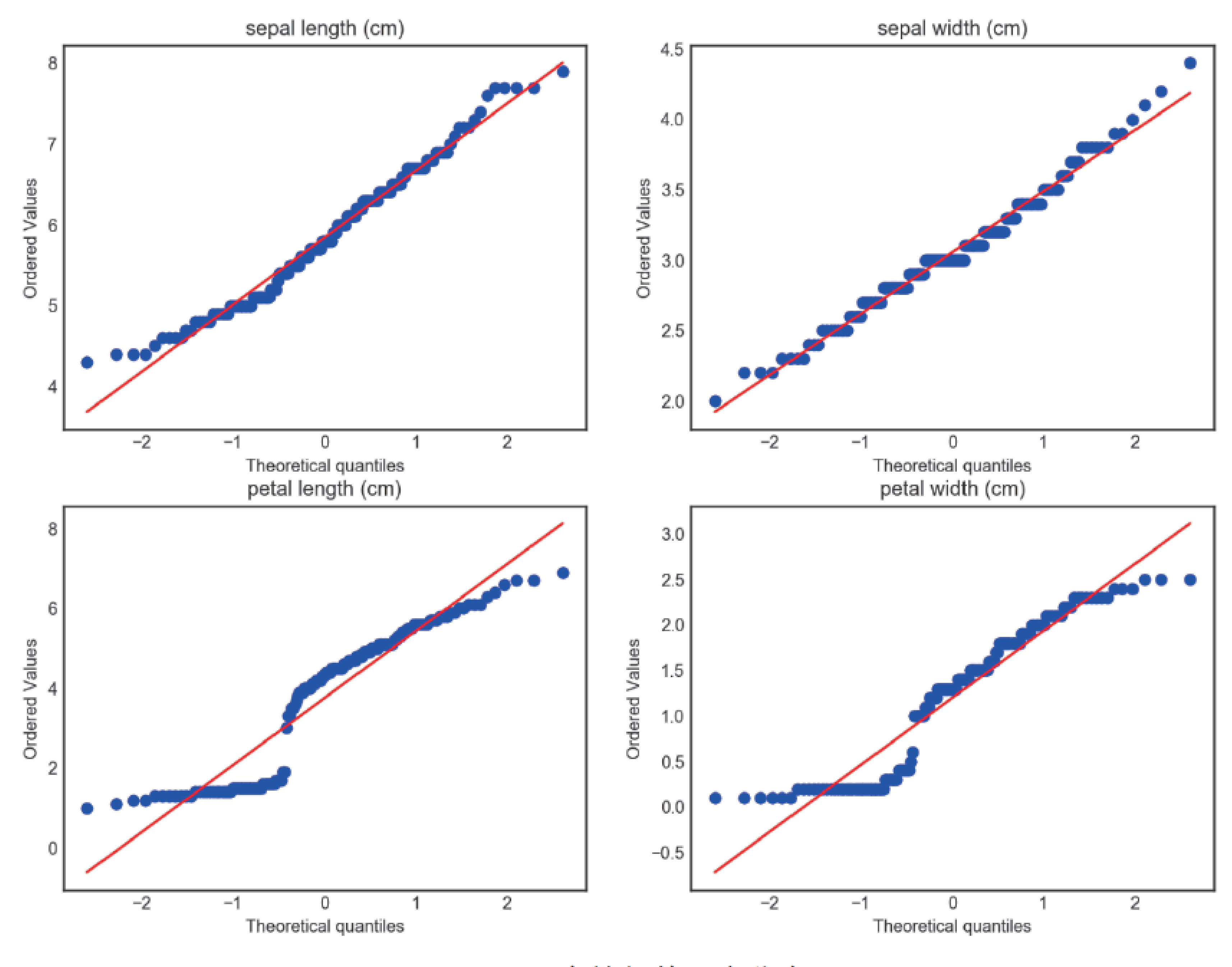


图 2-4 四个特征的正态分布

因此下一步的目标就是求具体每个分类的平均值、方差，如何做呢？这里当然可以提取不同分类对应的子矩阵，然后计算子矩阵的平均值、方差，读者可以自行尝试。我们这里提供另一种基于透视表（pivot_table）的方法。首先，对于 150×4 的二维矩阵特征，将其变为 600×1 的一维矩阵特征：

```
pd.melt(df, id_vars=['species'])
```

运行结果：

```
# out:
```

	species	variable	value
0	setosa	sepal length (cm)	5.1
1	setosa	sepal length (cm)	4.9
2	setosa	sepal length (cm)	4.7
3	setosa	sepal length (cm)	4.6
4	setosa	sepal length (cm)	5
5	setosa	sepal length (cm)	5.4
6	setosa	sepal length (cm)	4.6
7	setosa	sepal length (cm)	5
8	setosa	sepal length (cm)	4.4
9	setosa	sepal length (cm)	4.9

然后将这个一维矩阵特征通过透视表操作，针对每个特征值，以分类结果为行名称，求它们的平均值以及方差：

```
pd.melt(df, id_vars=['species']).\
        pivot_table(index=['species'], columns=['variable'],
                    aggfunc=[np.mean, np.var])
```

运行结果：

```
# out:
```

	mean				var			
	value				value			
Variable	petal length (cm)	petal width (cm)	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	sepal length (cm)	sepal width (cm)
species								
Setosa	1.464	0.244	5.006	3.418	0.030106	0.011494	0.124249	0.14518
versicolor	4.26	1.326	5.936	2.77	0.220816	0.039106	0.266433	0.098469
virginica	5.552	2.026	6.588	2.974	0.304588	0.075433	0.404343	0.104004

我们关注 petal length 和 petal width 两个特征，发现二者在 setosa 这个种类中要小于其他两个种类，并且 setosa 与 versicolor 之间的距离（4.260-1.464）也远高于三倍方差值（0.0301×3+0.2208×3），正好对应在图 2-4 中的两个间隔较远的正态分布。

继而我们以第三个特征为例，对其不同组分别进行正态分布检验：

```
fig = plt.figure(figsize=(12,4))

for i in range(3):
    name = data.target_names[i]
    ax = plt.subplot(1,3,i+1)
    stats.probplot(df[df['species']==name][data.feature_names[2]], plot=ax)
    ax.set_title(name)
```

其结果如图 2-5 所示。

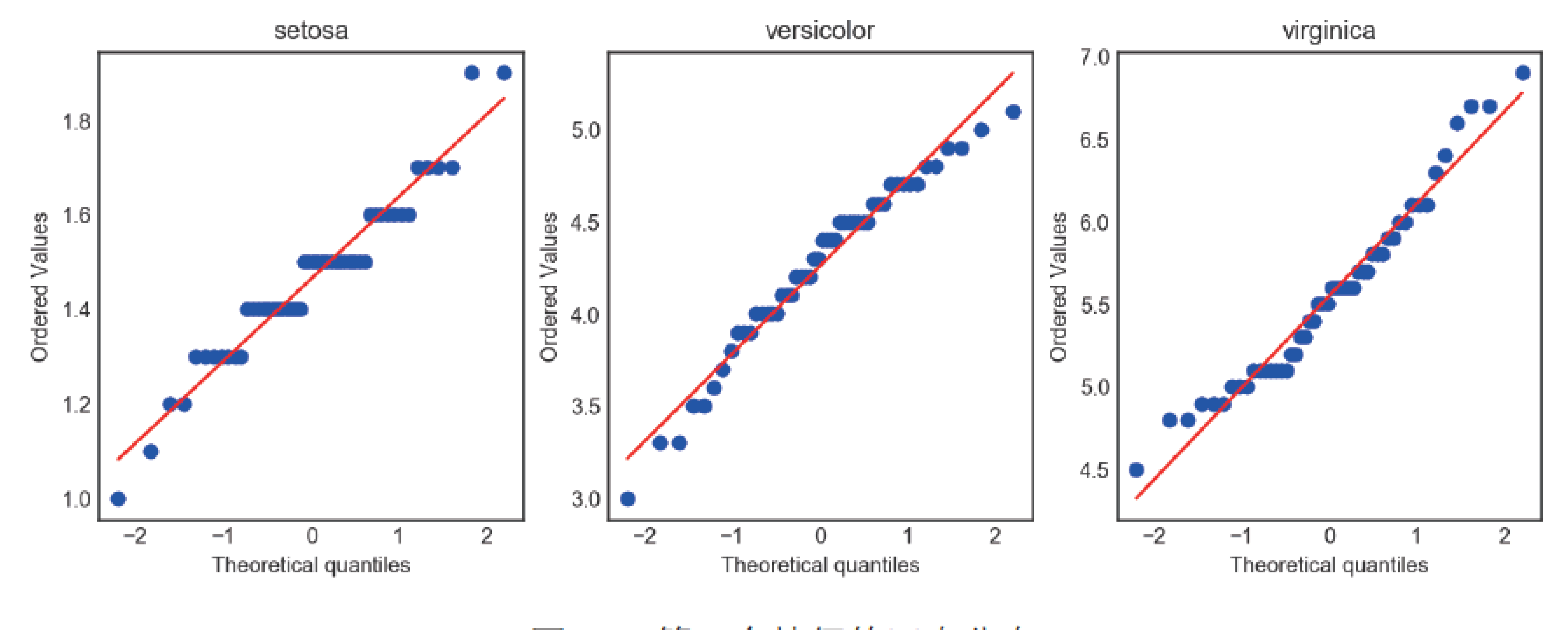


图 2-5　第三个特征的正态分布

由此可见，该特征在不同组内部是符合正态分布的。

能不能不建立模型，就看出三者的分类关系？

首先，这里说一下“不建立模型”的逻辑是什么。不建立模型就看出分类关系，是因为有时候分类的定义很简单，用复杂模型多少有点“杀鸡用牛刀”的意味。比如判断一个人是否酒驾，就一个指标，每百毫升血液里酒精含量是否大于 20mg，大于就是酒驾，否则不是，就是一个简单的 if…else 判断关系，不需要复杂的模型。实际工作中，如果找到了这种简单的 if…else 标准，其实是件好事，首先我们不需要费时费力地挖特征、训练模型，其次得出的结论也很简单，不涉及复杂的组合，也利于被人接受。

我们的鸢尾花数据集是否存在这种简单标准？可以借助简单的可视化技术来看一眼：

```
sns.pairplot(df, hue="species")
```

运行结果如图 2-6 所示。

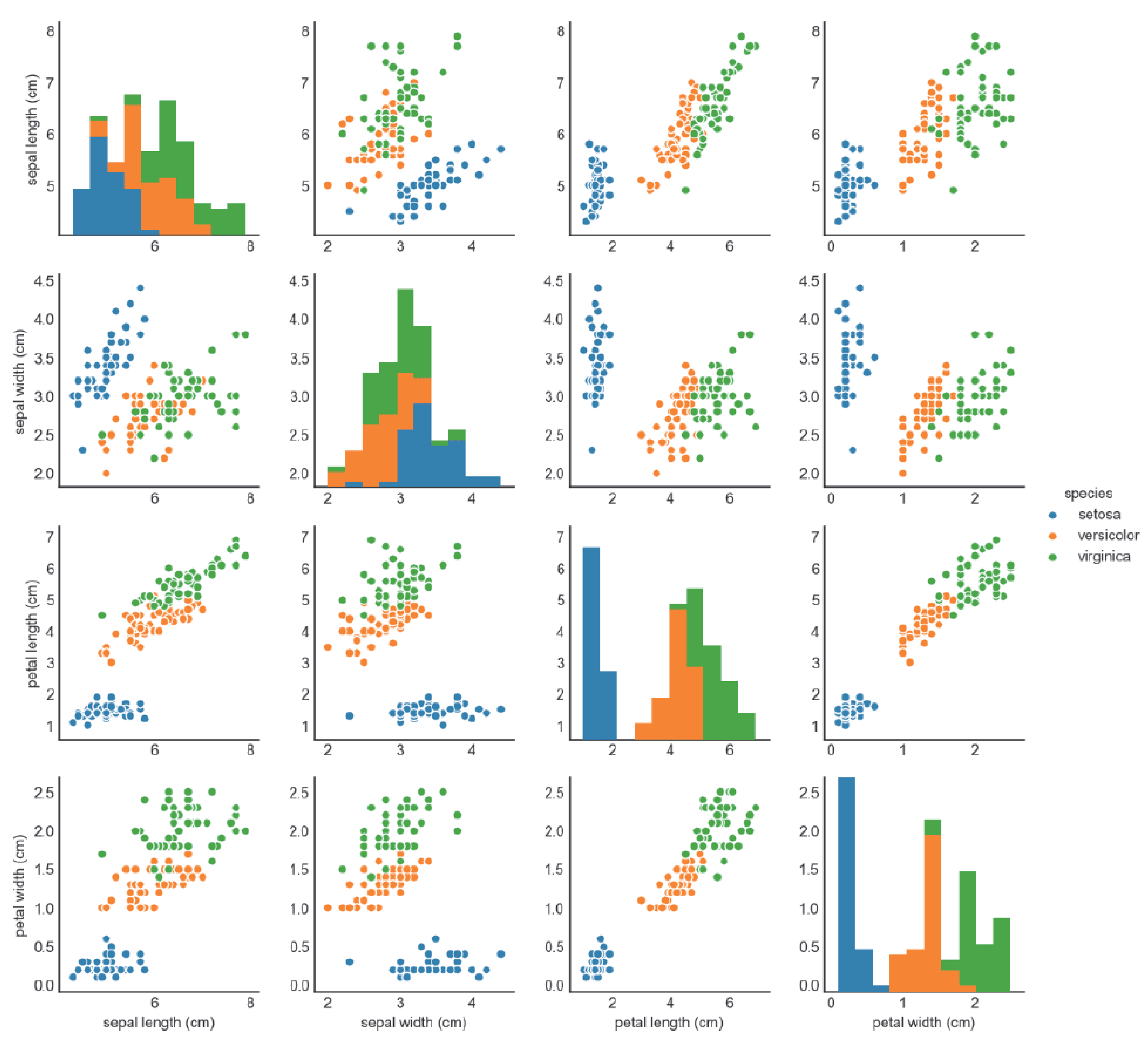


图 2-6 鸢尾花数据集的可视化

这个图包含了两层信息：第一层是单一值是否足以分开不同的分类结果，就是对角线上的四个分布图；第二层是数据间的两两组合是否足以分开不同的分类结果，就是非对角线上的散点图，其中左下角与右上角部分 x - y 坐标轴是对称的。

对于第一层信息，我们注意到第三行、第三列的点图中 setosa 与其他两类在 petal length 这个特征上是可以完全区分开的，即 petal length 小于 2 的是 setosa、大于 2 的是其他两种分类。另外两个种类区分得并不好，有很多的重叠部分。

对于第二层信息，根据点图判断的话，所有第三行、第三列的点图中，setosa 与其他两个种类也是可以完全区分开的。因为这些图是特征的两两组合，这些组合都使用第一层信息就可以明确区分 setosa 的特征——petal length，自然在两两组合中也可以区分。通过两两组合，另外两种也并非完全分开，但是重叠部分的交集有多有少——sepal 的长宽重合得多，petal 的长宽重合就少得多。

这时候读者会有新的问题了——**既然两个两个组合，看起来后两类重叠的面积少了很多，那么特征如果进行三个组合，画三维立体图重叠的是否会更少？**这里不太好画三维的点图，读者可以画四个三维的图看一眼。这里之所以不画并非是笔者懒，而是既然可以三个特征组合，就可以四个一起组合，这种情况下就没法画图了，毕竟我们生活在一个三维的宏观世界，要表示一个四维的坐标系难度其实挺大的。

因此，我们这里的结论是，如果想找出 setosa，那么不用建模也可以得出结论，花瓣（petal）长度小于 2cm 的就是。如果想找出其他两种，结论就不是肯定的了。

当然，对于这种直观方法的局限性，读者应该也发现了，就是特征两两组合会造成维度灾难。我们这里是四个特征，两两组合就会得到 $4\times3\div2=6$ 种不同的组合方式。如果是几百个特征，就是上万种组合方式，几万特征就是上亿种组合方式。这种情况下，要是再画这种组合图，人工找出特征组合就不现实了。我们需要计算机帮助组合特征，而计算机组合特征的方法就是借助机器学习模型。

对于各种机器学习模型，本书的篇幅主要是介绍监督学习方法，就是之前提到的应试教育。在此之前，我们先简单介绍下非监督学习找组合特征的方法，并且基于非监督学习的特征组合，对数据进行一个分类结果的预览。

能否对样本的分类情况进行简单的预览？

上文提到，直接将特征以两两组合的方式画在二维平面上，在特征过多时，会由于组合过多使得这种方法难以发挥作用。这时能否直接在二维平面上看到四维的分布情况呢？可以通过降维做到。

提到“降维”，最经典的一段文学描述莫过于《三体》这部科幻小说，在最后阶段描写的外星人使用“二向箔”，对太阳系进行的一场降维打击：

在二维化的太空艇上，可以看到二维展开后的三维构造，可以分辨出座舱和聚变发动机等部位，还有座舱中那个卷曲的人体。在另一个二维化的人体上，可以清楚地分辨出骨骼和脉络，也可以认出身体的各个部位。在二维化的过程中，三维物体上的每个点都按照精确的几何规则投射到二维平面上，以至于这个二维体成为三维太空艇和三维人体的两张最完整最精确的图纸，其所有的内部结构都在平面上排列出来，没有任何隐藏。

这是一段非常具有启发意义的描述。因为其实降维打击可以很容易，外星人完全可以用一个巨大的苍蝇拍直接拍扁整个太阳系，不过这样就做不到“其所有的内部结构都在平面上排列出来，没有任何隐藏”。可见**降维十分容易，降维同时尽可能地保留原有的结构难，**例如前面的 seaborn.pairplot 实际上就是进行了 12 次四维到二维的降维，每次都用了其中的两组特征，然后扔掉了剩下的两组。于是我们就思考如何尽可能多地在二维平面保留高维特征背后的信息。

主成分分析（PCA）是其中的一种思路。PCA 有两种通俗易懂的解释，一是最大化投影后数据的方差，即让数据在投影到的平面上更分散；二是最小化投影造成的损失，即让数据到投影平面的垂直距离最小。更通俗地说，就是 PCA 实际上是在寻找一个能把苍蝇在墙上拍得最扁、展开面积最大的一种角度。这种降维方法与二向箔给太阳系降维时使用的展示所有细节的黑科技降维方法当然没法比，但也非常有用，可以将其运用在鸢尾花的数据集上。

这种算法运用在鸢尾花数据集上的效果如下。需要读者的注意是，为了方便展示，这里用了鸢尾花数据集的前三个特征进行主成分分析，实际运用中需要展示全部特征。

```
    from sklearn.decomposition import PCA
    pca = PCA()
    df_sub = df[data.feature_names[0:3]]
    pca.fit(df_sub)

    pca_result = pca.transform(df_sub)
    fig = plt.figure(figsize=(4,4))
    ax = fig.add_subplot(111)
    ax.scatter(pca_result[:, 0], pca_result[:, 1], c=data.target, cmap=plt.
cm.Set3)
```

其运行结果如图 2-7 所示。

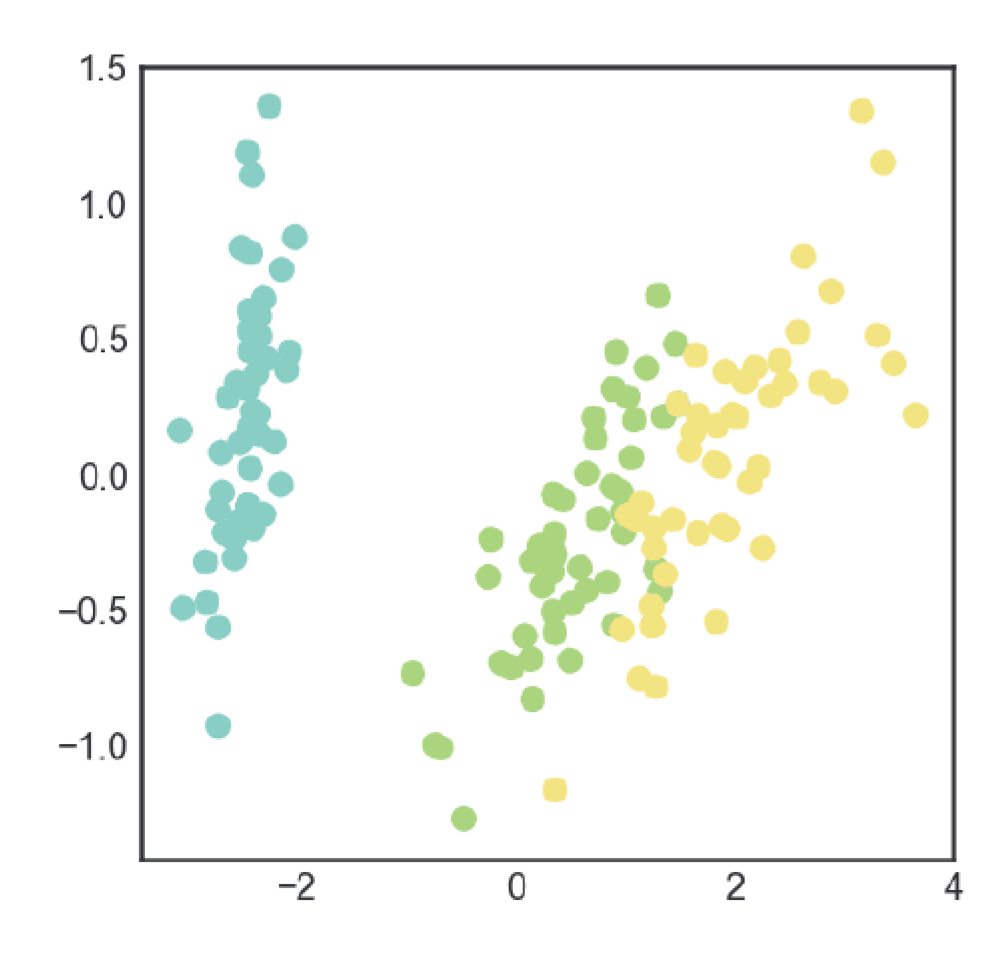


图 2-7 前三个特征进行主成分分析

如果希望知道投影平面是什么，可以通过协方差矩阵得到。

```
    from mpl_toolkits.mplot3d import Axes3D

    # 这两个参数用于调整投影平面的大小
    # 参考 sklearn 范例 http://scikit-learn.org/stable/auto_examples/
decomposition/plot_pca_3d.html

    plane_show_size_ratio = 5
    plane_show_shift = df_sub.mean().values
    pca_score = pca.explained_variance_ratio_
    V = pca.components_
    l_pca_axis = V.T * plane_show_size_ratio
    l_pca_plane = []
    for pca_axis in l_pca_axis:
        l_pca_plane.append(np.r_[pca_axis[:2], - pca_axis[1::-1]].
reshape(2,2))

    fig = plt.figure(figsize=(4,4))
    # 画点
    ax = Axes3D(fig, rect=[0, 0, .95, 1], elev=150, azim=-34 )
    ax.scatter(df_sub.values[:,0], df_sub.values[:,1],df_sub.values[:,2], '.',
c=data.target, cmap=plt.cm.Set3)

    # 根据 PCA 结果旋转 3d 图形，使之达到 " 最大化投影后数据的方差，即让数据在投影到的平面
上更分散；二是最小化投影造成的损失，即让数据到投影平面的垂直距离最小 " 的效果
    ax.plot_surface(l_pca_plane[0]+plane_show_shift[0],
                    l_pca_plane[1]+plane_show_shift[1],
             l_pca_plane[2]+plane_show_shift[2], alpha=0.1)
```

其运行结果如图 2-8 所示。

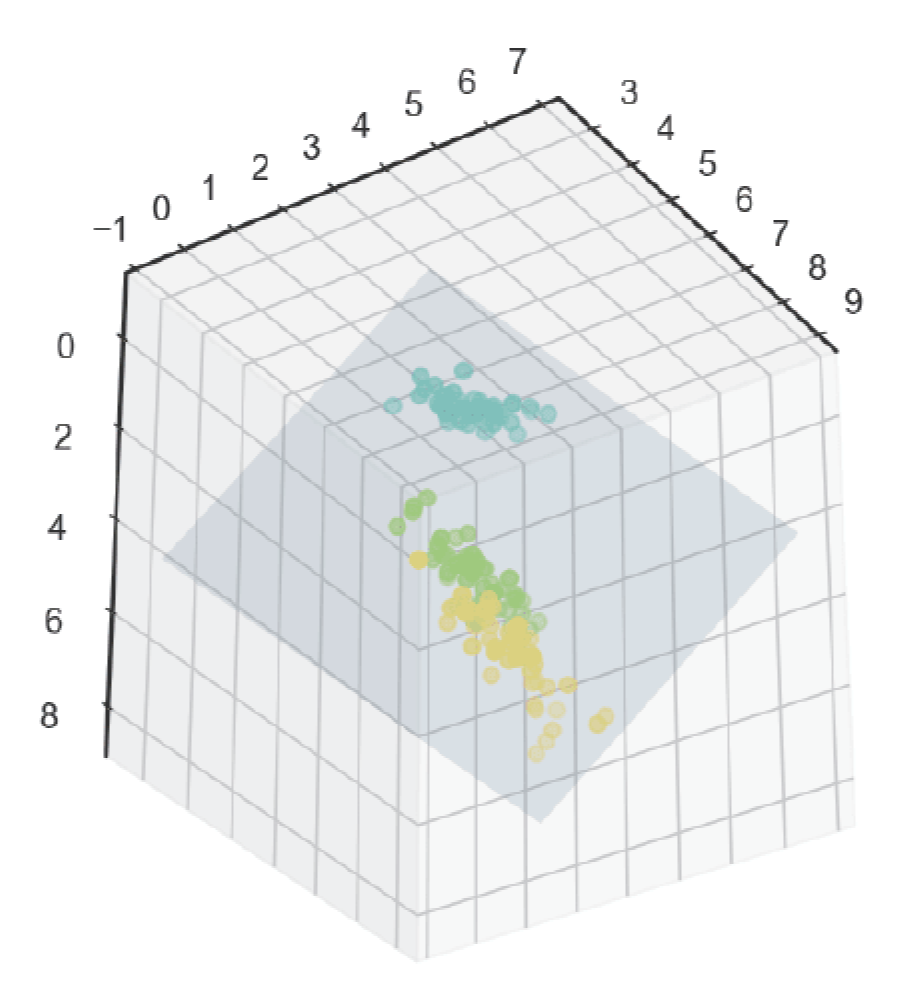


图 2-8　通过协方差矩阵得到投影平面

由此可见，这里找到的投影平面是数据展开效果非常好的一个平面，符合 PCA 定义的预期。

最后，请读者思考 PCA 的定义。首先，投影平面是否必须是一个平面？能否是一个曲面，比如广义相对论里那种扭曲的空间。其次，这里“最大化投影后数据的方差”的目标，是否可以修改？答案当然是肯定的，这部分的详细介绍请进一步学习“流形学习”（manifold learning）相关的内容。这些方法在我们的数据集上的简单使用效果如下：

```
    from sklearn.manifold import Isomap, MDS, SpectralEmbedding
    n_components = 2
    n_neighbors = 10
    X = df.drop(['species'], axis=1)
    color = data.target
    fig = plt.figure(figsize=(12, 4))

    Y = Isomap(n_neighbors, n_components).fit_transform(X)
    ax = fig.add_subplot(131)
    ax.scatter(Y[:, 0], Y[:, 1], c=color, cmap=plt.cm.Set3)
    ax.set_title("Isomap")

    Y = MDS(n_components, max_iter=100, n_init=1).fit_transform(X)
    ax = fig.add_subplot(132)
    ax.scatter(Y[:, 0], Y[:, 1], c=color, cmap=plt.cm.Set3)
    ax.set_title("MDS")

    Y = SpectralEmbedding(n_components=n_components,n_neighbors=n_
neighbors).fit_transform(X)
    ax = fig.add_subplot(133)
    ax.scatter(Y[:, 0], Y[:, 1], c=color, cmap=plt.cm.Set3)
    ax.set_title("Isomap")
```

其运行结果如图 2-9 所示。

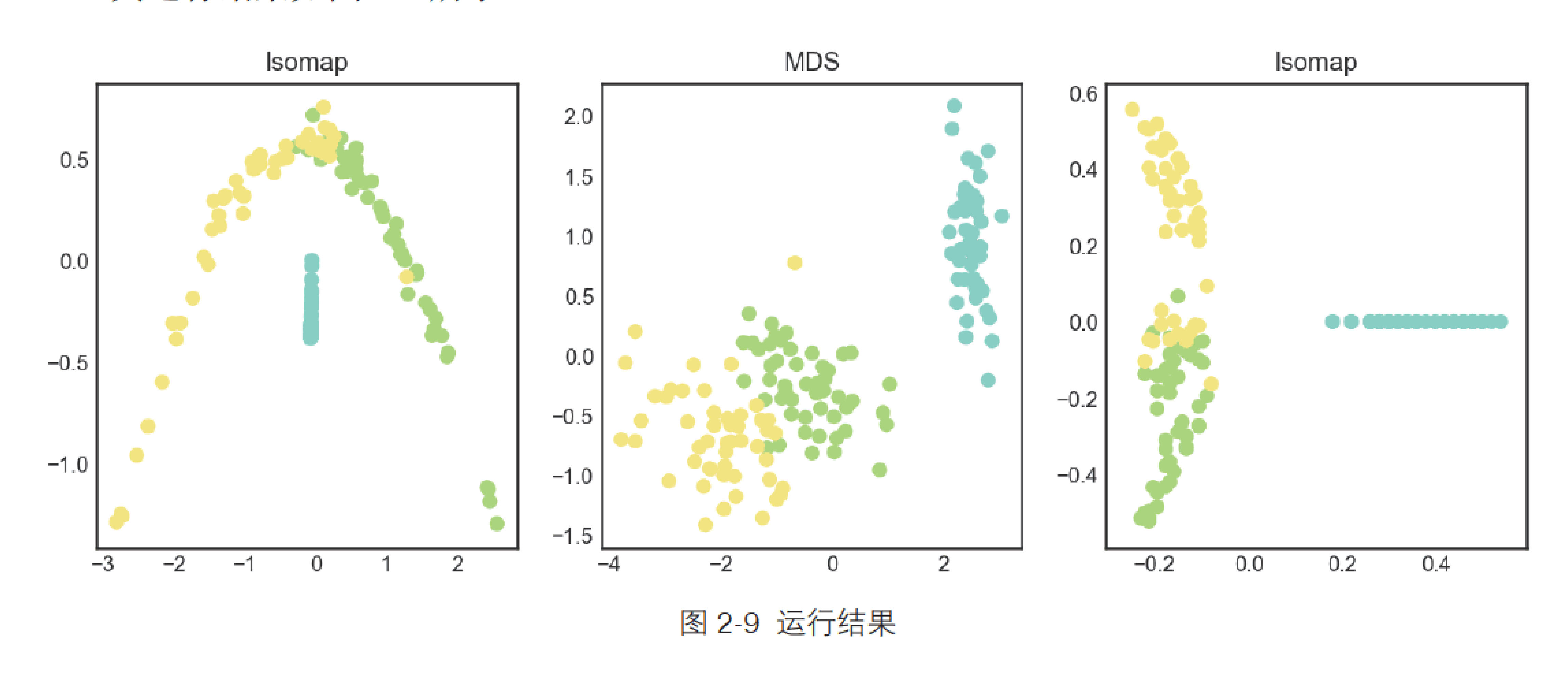


图 2-9 运行结果

2.3 数据挖掘与训练模型

我们在前面预览环节中，通过特征两两组合展示，以及主成分降维的方法，已经对数据有了初步的认识。但是这种认识，不等于明确的分类标准，真正用于实战中的模型，还是需要使用监督学习方法。接下来的内容，将介绍监督学习的一个基本流程。

2.3.1 第一步，准备数据

这一部分包括两部分的内容，一是将所有数据划分训练集、验证集与测试集，二是进行数据的标准化。

首先解释一些名词，即什么是训练集、验证集与测试集。如同前面提到的，监督学习就是一种唯分数论的应试教育，准确率越高越好，正如学生考试成绩越高越好一样。那么提高应试教育考试成绩的方法，同样也可以用在监督学习上，那就是题海战术。在题海战术的过程中，老师会拿一堆卷子给学生做，有的作为家庭作业，有的作为阶段性考试，如周考、月考等，最后留一些题目用在期末考试中，参见表 2-3。

我们不妨把机器学习模型理解成做题的学生。平时学生做作业时是允许参考题目答案的，这也有助于学生理解解题思路；而**有监督的机器学习模型在训练过程中，同样需要一份“题目”和“答案”**，比如鸢尾花数据集中，150 个样本就相当于 150 个考题，每个题目给出四个特征，要模型预测分类结果，参考答案这里也一并给出，让模型在训练阶段中不断地“对答案”，训练高分模型。

当然，这样做有一个问题，学生如果直接抄答案，作业会完成得又快又好，如何让这部分学生现行？考试，考试题不附有参考答案，抄作业的就现行了，所以老师会不断地进行小范围阶段性考试，检验学生的学习效果；有监督的机器学习也是一样，会使用验证集检验学生的学习效果，确认模型对于未知数据也有很好的表现。

最后，阶段测试考得再好，期末考试或者高考这样的大型考试没考好也没有用。这种大型考试不是给学生来学习的，而是用来排名比较的；机器学习最终的模型好不好，也需要看它在测试集上的表现，抛开其他一切因素，99.95% 的准确率就是比 99.94% 的准确率要好。

表2-3 将机器学习模型理解为做题的学生

题海战术	机器学习	特　点
家庭作业	训练集（Training Set）	同时有题目和答案，可以用来优化模型
阶段考试	验证集（Validation Set）	看题目对答案，不参与模型训练，只用来检验模型准确率，进而指导人工的调参优化
期末考试	测试集（Testing Set）	看题目给结果，给模型最终打分，是不同模型好坏排名的依据

通常情况下，数据的所有者在拿到完整数据后会进行第一次划分，分出初步训练集以及测试集，将初步训练集的数据和结果以及测试集的数据交给数据科学家，然后自己留下测试集的分类结果，将用于评价不同数据科学家、不同模型提交的准确率。数据科学家拿到初步训练集

后进行第二次划分，分出训练集以及验证集，用于训练模型以及对模型进行自我评价。

我们以题海战术为例，让大家更容易理解数据划分的目的，继而在收集数据、划分数据时要做到符合以下常识：

（1）题目和答案要有关系。同样，收集到的数据特征也必须和数据要预测的东西有关系。当然，这里并不要求所有数据都有关系，可以有冗余，也可以有干扰项。确认这一点的一种简单方法是，将这些数据给这个领域的人类专家，如果他可以对结果做出判断，那么机器就可以。

（2）合理划分作业和测试的比例。比如做单选题，如果只选 ABCD 后对答案，不思考背后的原因，可能做一份卷子和做一百份卷子没有什么区别，答案对多了还可能会得出“三长一短选最长”这种结论。同样，只做作业不考试，可能会在学习方法上产生方向性的错误。机器学习也一样，手里拿到一些数据之后，不妨将 70% 拿来作为训练集（家庭作业），30% 拿来作为验证集（小测）。题海战术的题目，要尽量保证不同题型在作业和考试中一致，同样，不同种类的数据，在训练集验证集中的比例也应当保持一致。

（3）避免题目泄露。测试集中的数据不应出现在训练集和验证集中。

数据科学性除了需要合理进行数据集的划分之外，还需要对数据进行**标准化操作**。具体而言，就是很多样本通过减平均值、除标准差的方法将数据变成标准正态分布。这种做法的主要原因是，在训练数据的过程中，模型的参数会不断调整，而调整过程中，同样调整 100，如果特征 A 的平均值是 1，这个调整幅度就会显得过大，而对于平均值是 10000 的特征 B 而言，这个调整幅度就会显得过小，模型会浪费大量时间去适应不同特征的分布，从而影响训练的收敛速度。因此如果这里统一成平均值是 0、方差为 1 的标准正态分布，会减小训练开销，得到更好的训练结果。这一点在深度学习图像处理中尤为关键，因为图像的像素值大小是 0~255，如果直接使用则距离计算机喜欢的标准正态分布有些差距，所以通常会将像素范围调整在 [-1, 1] 之间，或者直接处理成标准正态分布。

这里需要注意，在实际处理过程中，初学者容易忽略的一点是对训练集、验证集、测试集分别计算不同的平均值与标准差，然后分别减平均值除标准差。这里应该**统一减去训练集的平均值和标准差**，因为首先我们可以认为是从训练集中估计了整体分布的特点，其次这样做也避免了引入更多验证数据后平均值标准差变化造成的影响。

最后我们放上 sklearn 在这一部分的用法：

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# df 作为初步训练集，划分 80% 作为训练集、20% 作为验证集
df_train, df_val = train_test_split(df, train_size=0.8, random_state=0)

# 提取特征，这里是将分类结果舍弃
X_train = df_train.drop(['species'], axis=1)
X_val   = df_val.drop(['species'], axis=1)
# 提取分类结果
```

```
    Y_train = df_train['species']
    Y_val   = df_val['species']

    # 设定分布 X_scaler，用训练集估计（fit）分布，然后对验证集进行转换（transform）
    X_scaler = StandardScaler()
    X_trainT = X_scaler.fit_transform(X_train)
    X_valT   = X_scaler.transform(X_val)
    # 这里将保证训练集是标准正态分布，验证集不一定满足这个条件，但不会差很多
    print(X_trainT.mean(axis=0), X_trainT.var(axis=0))
    # [  0.00000000e+00  -7.49863135e-16   4.25585493e-16   1.22124533e-16]
[ 1.  1.  1.  1.]
    print(X_valT.mean(axis=0), X_valT.var(axis=0))
    # [-0.22139933  0.00775008 -0.16081079 -0.20798778] [ 0.70916187
1.04757042  0.87342285  0.8028885 ]
```

2.3.2 第二步，挖掘数据特征

特征工程是整个建模过程的重中之重，通常建立一个数学模型， 70% 以上的时间都是投入在数据挖掘环节中的。

我们挖掘特征的原因是为了让计算机可以更好地理解数据。要想让计算机理解数据，数据科学家就需要首先理解数据。这种理解，在实际运用的过程中是离不开具体业务逻辑支持的。例如，医学领域的统计学家统计了各种疗法对降低血糖的效果，希望建立一个推荐高血糖治疗方法的模型，最后发现打胰岛素见效快、效果好。而在实际的医学临床实践中，打胰岛素是最迫不得已的一种治疗方法，如果有更好的选择，医生绝对不会推荐这种方法。因此，虽然现阶段人工智能不断地在各个领域取得很好的表现，但是如果想要在复杂的应用场景落地，还是需要有人类专家的支持与帮助。

正是由于这一部分内容涉及面太广，不属于入门内容，并且我们后文的深度学习部分主要强调的是用深度神经网络自动挖掘特征，因此这部分内容将用最简单的例子讲一讲。

在二维平面上以原点为圆心，在两个圆环的范围分别随机生成与原点距离不同的两组点。其中里面圆环上的点是第二组，外面圆环的点是第一组。我们看看如何挖出一个简单的、线性可分的特征来区分这两组点：

```
    from sklearn.utils import shuffle
    import matplotlib as mpl
    from cycler import cycler
    mpl.rcParams['axes.prop_cycle'] = cycler(color='rb')
    np.random.seed(42)
    pseudoNum1 = 300
    pseudoNum2 = 300
    np_pho1    = 4.5 + np.random.rand(pseudoNum1)*2
```

```
    np_pho2   = 0.5 + np.random.rand(pseudoNum2)*2
    np_theta1 = np.random.rand(pseudoNum1)*360 / 2*np.pi
    np_theta2 = np.random.rand(pseudoNum2)*360 / 2*np.pi

    np_x1 = np_pho1 * np.cos(np_theta1)
    np_y1 = np_pho1 * np.sin(np_theta1)
    np_x2 = np_pho2 * np.cos(np_theta2)
    np_y2 = np_pho2 * np.sin(np_theta2)

    pd_circ = shuffle(pd.DataFrame({
        "X" : list(np_x1)+list(np_x2),
        "Y" : list(np_y1)+list(np_y2),
        "label" : ["Class1" for x in range(pseudoNum1)] + ["Class2" for x in
range(pseudoNum2)]
    }), random_state=0).reset_index().drop(['index'],axis=1)
    pd_circ0 = pd_circ.copy()
    pd_circ.head()
```

	X	Y	label
0	0.596878	-0.30041	Class2
1	5.112182	-0.49413	Class1
2	-3.34097	3.760705	Class1
3	-1.54582	0.033936	Class2
4	-0.84614	4.438461	Class1

看图应该更加直观：

```
    for sub in ["Class1", "Class2"]:
        pd_sub = pd_circ[pd_circ['label']==sub]
        plt.plot(pd_sub["X"], pd_sub["Y"], ".", label=sub)

    plt.legend()
```

其结果如图 2-10 所示。

对特征组合做图发现，如果使用单一特征的话，二者是混合在一起的。两个特征直接进行组合的话，二者之间仍然线性不可分——两个分类被一个圆形隔开了，一条线性的直线切不开两者。

```
    sns.pairplot(pd_circ, hue="label")
```

其运行结果如图 2-11 所示。

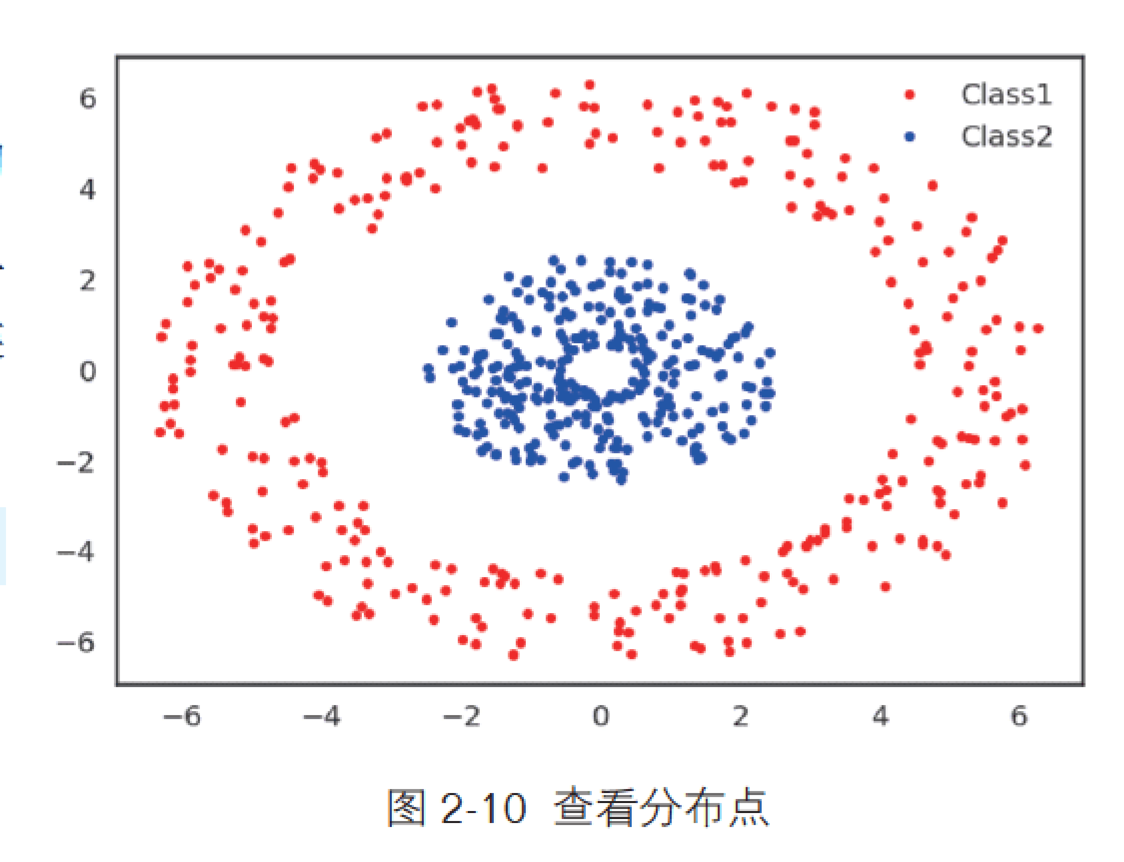


图 2-10 查看分布点

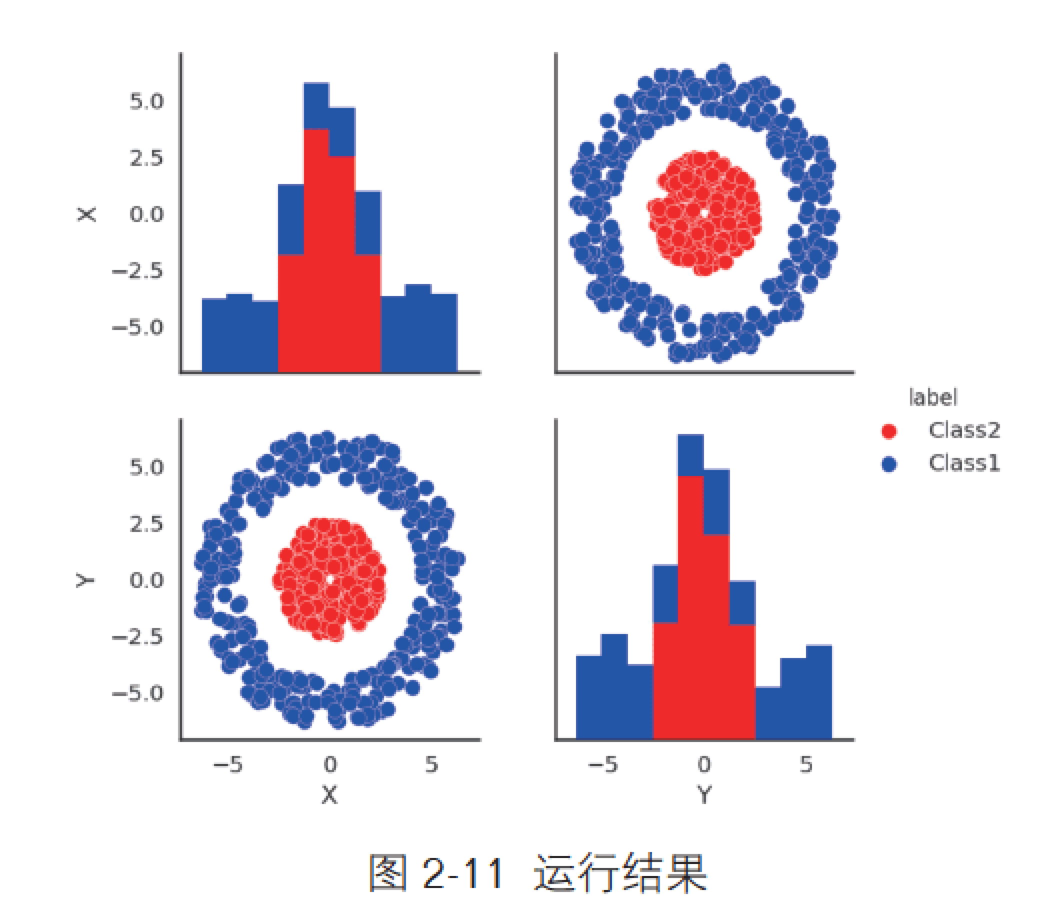


图 2-11　运行结果

这时可以手动进行一些特征工程。简单试一下加法和乘法（X+Y、X*Y），由于和圆形有关，我们再试下 X*X+Y*Y：

```
    pd_circ['X_add_Y'] = pd_circ['X'] + pd_circ['Y']
    pd_circ['X_time_Y'] = pd_circ['X'] * pd_circ['Y']
    pd_circ['X2_add_Y2'] = pd_circ['X'] * pd_circ['X'] + pd_circ['Y'] * pd_
circ['Y']
    sns.pairplot(pd_circ, hue="label")
```

其运行结果如图 2-12 所示。

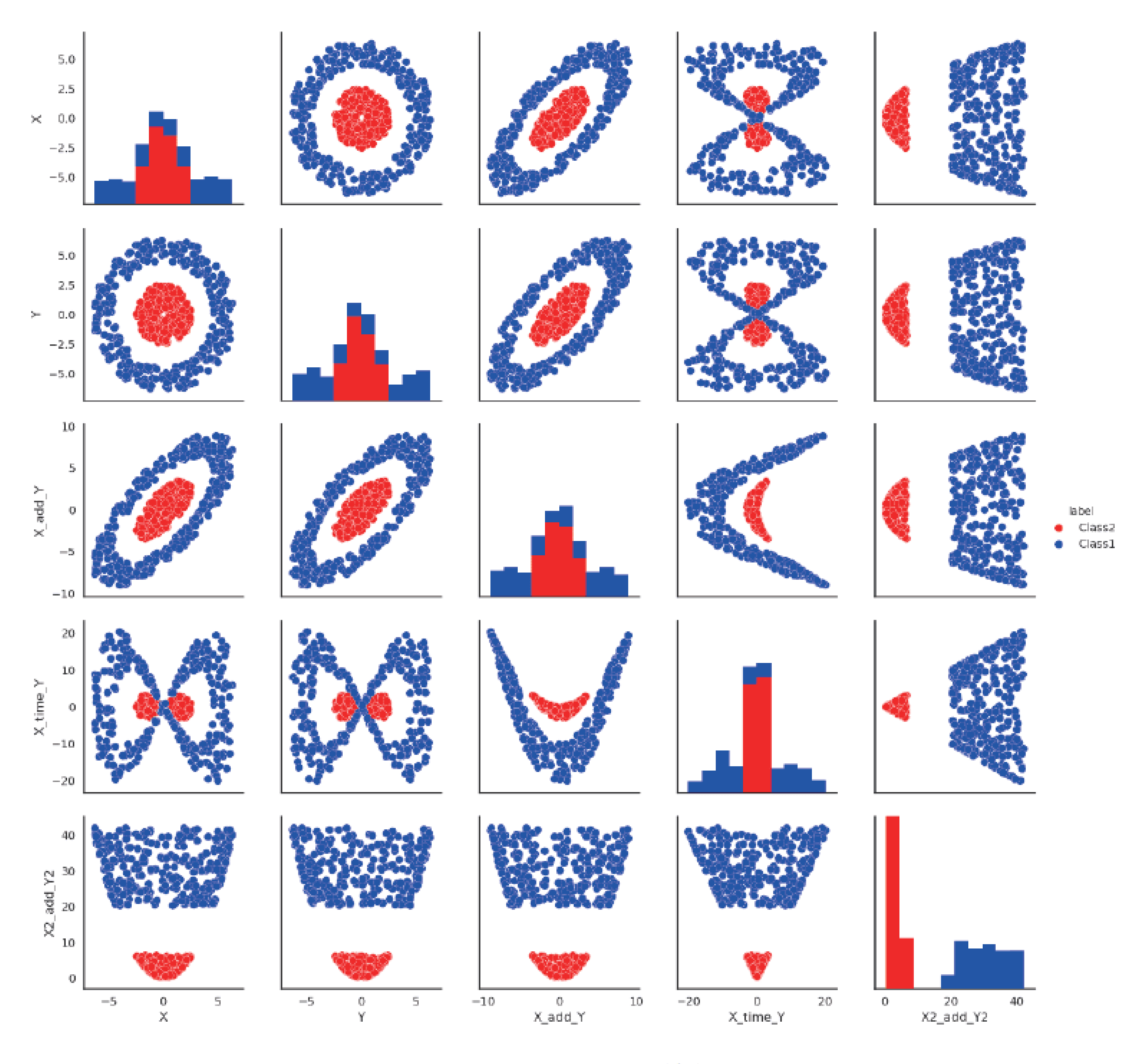


图 2-12　X*X+Y*Y 的结果

注意右下角的图会发现，我们挖掘的 X*X+Y*Y 这个特征，单独一个特征已经是线性可分的了，这极大地降低了计算机模型将二者分开的难度。

读者可能会问，难道计算机只认识线性可分、不认识圆形边界吗？如果有成百上千个特征，难道数据科学家在成百上千个特征基础上手动组合各种可能性？实际上计算机当然可以认出圆形的边界，这里只是很简单的特征工程例子，计算机同样可以借助数学模型实现。比如这里圆形边界的例子，就可以通过计算两个类群的高斯分布特征，进而借助高斯核函数实现分割：

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

```
pd_circ_melt = pd_circ0.melt(id_vars=['label']).\
   pivot_table(index=['variable'], columns=['label'],
   aggfunc=[np.mean, np.var])

mean_X = pd_circ_melt['mean']['value'].loc['X'].values.reshape(2,1)
mean_Y = pd_circ_melt['mean']['value'].loc['Y'].values.reshape(2,1)
var_X  = pd_circ_melt['var']['value'].loc['X'].values.reshape(2,1)
var_Y  = pd_circ_melt['var']['value'].loc['Y'].values.reshape(2,1)

probX = 1 / np.sqrt(2*np.pi*var_X) * \
   np.exp(-1*(np.array([pd_circ['X'].values,pd_circ['X'].values]).\
   reshape(2,600)-mean_X)**2 / (2*var_X))

probY = 1 / np.sqrt(2*np.pi*var_Y) * \
   np.exp(-1*(np.array([pd_circ['Y'].values,pd_circ['Y'].values]).\
   reshape(2,600)-mean_Y)**2 / (2*var_Y))

pd2 = pd.DataFrame(probX.T*probY.T)
pd_circ['pred'] = pd2.apply(lambda x: "Class2" if x[0] < x[1] else
"Class1", axis=1)
pd_circ
```

	X	Y	label	X_add_Y	X_time_Y	X2_add_Y2	pred
0	0.596878	-0.30041	Class2	0.296467	-0.17931	0.446509	Class2
1	5.112182	-0.49413	Class1	4.618049	-2.5261	26.37857	Class1
2	-3.34097	3.760705	Class1	0.419736	-12.5644	25.30497	Class1
3	-1.54582	0.033936	Class2	-1.51189	-0.05246	2.390721	Class2
4	-0.84614	4.438461	Class1	3.592322	-3.75555	20.41589	Class1

以高斯分布来拟和各个类群 XY 的分布，继而通过高斯分布给出的概率得出的分类结果是 pred 列，而 pred 列中给出的结果与实际结果是基本符合的。可见借助算法模型，计算机可以实现多个特征的组合。但是这种组合往往缺乏针对性，如果人类专家能在此手动挖掘几个关键特征，

“提示”计算机一下，将更加可能得到好的结果。比如计算机并不直接认识文本、年月日、经纬度、人物关系等，需要人类专家在现有数据基础上，根据需要做一些基本转换，这些信息才可以被有效利用。

最后说两点在实际特征工程进行中入门者需要注意的事项：

（1）对于离散特征，使用 one-hot 编码。

例如，对于这种输入：

name	job
张三	工人
李四	农民
王五	军人
李武	工人

我们想用数字来表示职业。很多初学者会用这种错误的方法来表示：

name	Job	job_id
张三	工人	1
李四	农民	2
王五	军人	3
李武	工人	1

这种方法错误的原因是，**如果用连续的数字来表示不同职业，计算机会认为这些数字存在大小关系，由于 1 < 2 < 3，因此工人 < 农民 < 军人，这种观点当然是错误的，**所以这里相当于是给了模型一个经过错误处理的输入数据。对于这种情况，我们使用 one-hot 编码来表示。如果用 one-hot，这里正确的表示方法就是某个人是不是工人、是不是农民、是不是军人：

```
import pandas as pd

df_onehot_example = pd.DataFrame({
   "name"  : [u"张三", u"李四", u"王五", u"李武"],
"job"  : [u"工人", u"农民", u"军人", u"工人"]
})

pd.get_dummies(df_onehot_example, columns=["job"])
```

	name	job_军人	job_农民	job_工人
0	张三	0	0	1
1	李四	0	1	0
2	王五	1	0	0
3	李武	0	0	1

（2）特征工程其实是个头脑风暴的过程。在开始阶段，特征要尽可能多，到了后期，则要尽可能地选择最重要的特征用于模型训练。这种选择可以通过在模型中引入正则化来完成，至于多保留特征还是多舍弃特征，这里可以通过调整正则化常数来实现，调整结果的好坏可以进一步在验证集中的表现来确认。

初学者可能会忽略这一点，看见模型预测准确率在训练集中很高，一看准确率 99% 就以为训练成功，忘记在验证集中确认模型的表现。如果此时验证集的表现并不好，数据就发生了**过拟合**现象。我们用应试教育的例子来类比的话，一个学生平时作业准确率很高，他的学习方法就是“背答案”，而且背得很准，见过的题目全部都知道答案，但是题目稍微变一下，就不会做题了，于是考试时拿到新题目，准确率就下来了。这种现象归根结底是无法很好地适应未知情况。

同样，用监督学习的名词来替换应试教育，见过的数据都能预测对，没见过的准确率大幅降低，这种情况也是由于数据不能合理地适应未知情况。这种模型并不是我们需要的结果。

最后，过拟合的前提是，学生平时成绩很好，考试没发挥好。如果平时成绩就不好，考试也没考好，这种情况属于**欠拟合**。如果发生欠拟合，引入正则化就不是那么紧迫了，我们应该更多关注关键特征是否被正确挖掘、模型是否合理、数据是否充足等。我们举一个多项式回归的例子，图中是在用多项式拟合加入噪声的 cos 曲线的一部分，左图是一次多项式拟合，显然由于一次多项式是线性的，拟合曲线肯定不够合理，于是发生了欠拟合；中间的使用 4 次多项式，看起来不错；而右边的十次多项式拟合十个点，结果看起来误差更小、更完美。但是假如再从 cos 函数中抽取若干点，这条线拟合的效果就会差很多，因此属于过拟合的情况，需要用正则化减少多项式的次数。有关正则化具体如何实现，下一部分讲解使用模型时将进一步介绍。

```
    # 引用自 http://scikit-learn.org/stable/auto_examples/model_selection/
plot_underfitting_overfitting.html
    import numpy as np
    import matplotlib.pyplot as plt
    from sklearn.pipeline import Pipeline
    from sklearn.preprocessing import PolynomialFeatures
    from sklearn.linear_model import LinearRegression
    from sklearn.model_selection import cross_val_score
    %matplotlib inline

    def true_fun(X):
        return np.cos(1.5 * np.pi * X)

    np.random.seed(42)

    n_samples = 30
    degrees = [1, 4, 15]

    X = np.sort(np.random.rand(n_samples))
    y = true_fun(X) + np.random.randn(n_samples) * 0.1

    plt.figure(figsize=(14, 5))
    for i in range(len(degrees)):
        ax = plt.subplot(1, len(degrees), i + 1)
        plt.setp(ax, xticks=(), yticks=())

        polynomial_features = PolynomialFeatures(degree=degrees[i],
                                                 include_bias=False)
```

```
        linear_regression = LinearRegression()
    pipeline = Pipeline([("polynomial_features",
    polynomial_features), ("linear_regression", linear_regression)]
       )
        pipeline.fit(X[:, np.newaxis], y)
        # Evaluate the models using crossvalidation
        scores = cross_val_score(pipeline, X[:, np.newaxis], y,
                          scoring="neg_mean_squared_error", cv=10)

        X_test = np.linspace(0, 1, 100)
        plt.plot(X_test, pipeline.predict(X_test[:, np.newaxis]),
label="Model")
        plt.plot(X_test, true_fun(X_test), label="True function")
        plt.scatter(X, y, edgecolor='b', s=20, label="Samples")
        plt.xlabel("x")
        plt.ylabel("y")
        plt.xlim((0, 1))
        plt.ylim((-2, 2))
        plt.legend(loc="best")
        plt.title("Degree {}\nMSE = {:.2e}(+/- {:.2e})".format(
            degrees[i], -scores.mean(), scores.std()))
    plt.show()
```

其运行结果如图 2-13 所示。正则化系数过高（左）、过低（右）都会影响拟合的效果。

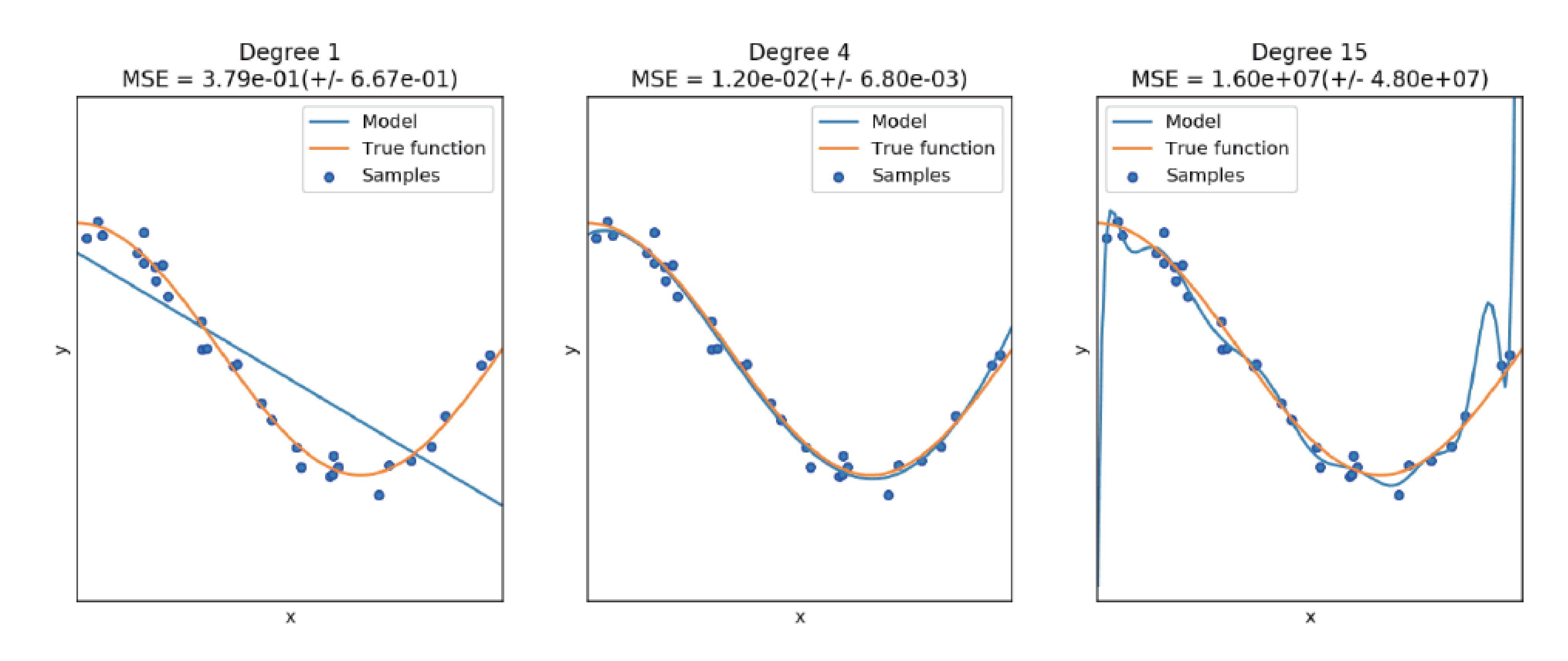


图 2-13　多项式曲线拟合过程中引入正则化

2.3.3 第三步，使用模型

这一步就是大家通常所说的“调（diao）包”以及“调（tiao）参”了，即调用算法包中的现有模型，通过调整模型参数，使模型在数据集中得到较好的表现。本书主要介绍的深度神经网络也是一种模型，后文将会详细介绍这一点。

经常有人说机器学习从业人员就是在调包和调参，这个观点其实是值得商榷的。因为首先如上文所述，有经验的数据科学家在拿到数据后，主要的精力是放在数据挖掘上，再进一步说，如果考虑数据的收集环节，那么数据科学家大部分精力并不是放在模型上，而是放在数据的收集、整理、清洗环节上。

给这种现象再深挖一层原因，这个问题其实出在监督学习有明确的评价标准上。由于标准明确，因此机器学习就有了套路，然后这个套路就被理解成了调包、调参。这个标准就是模型给出的结果和实际结果是否一致。这是一个很好理解的标准，比如老师要培养学生的个人品质、人生观、价值观，可能并不是一件容易的事情，如何培养一句话说不清楚；如果老师要提高学生考试成绩，就相对容易了许多，因为考试有明确的范围以及评价标准，只需要关注这部分内容对症下药即可，如同机器学习从业人员为了提高模型表现，使用不同模型、调整不同参数一样，虽然不好做，终归还是有套路的；如果要让模型打游戏，模型就需要对游戏中各种行为的收益做一个衡量，此时模型也没有公认明确的量化指标，而是上升到“人生观、价值观”的层面了，评价模型就如同评价学生个人品质一样，同样也是一句话说不清楚的。

这里将从评价标准讲起，从后往前简单说一下这部分内容。首先简单介绍逻辑回归的公式，进而根据评价标准讲一下模型的损失函数问题，最后说一下上一步的正则化如何影响这里的损失函数。由于本书的入门性质，我们仍然选择简单的讲，这里主要介绍逻辑回归模型。

1. 逻辑回归的公式表达

虽然名字带了回归，但事实上，这里逻辑回归是一个有监督的分类问题。我们具体看一下逻辑回归做了什么。

接下来，本书将推出一些公式。初学者可能最怕的就是满篇的公式，为了方便读者理解，这里从高中课本开始说起。高中课本中有一个“一元二次回归”，形式如下：

$$y = ax + b$$

注意，高中课本上的 x 和 y 是一个数字，而实际上这里的 x 可以是一个长度是 m 的向量（vector）。背后的数学意义是，之前预测 y 只考虑一个因素，这里考虑 m 个因素，而这里的 m 个因素就可以是鸢尾花数据集中的四个特征（$m = 4$）：

$$y = a_1 x_d + a_2 x_2 + \cdots + a_m x_m + b$$

这里的公式用向量的形式重新定义：

$$\boldsymbol{y} = \boldsymbol{\omega}^{\mathrm{T}} \boldsymbol{x}$$

其中

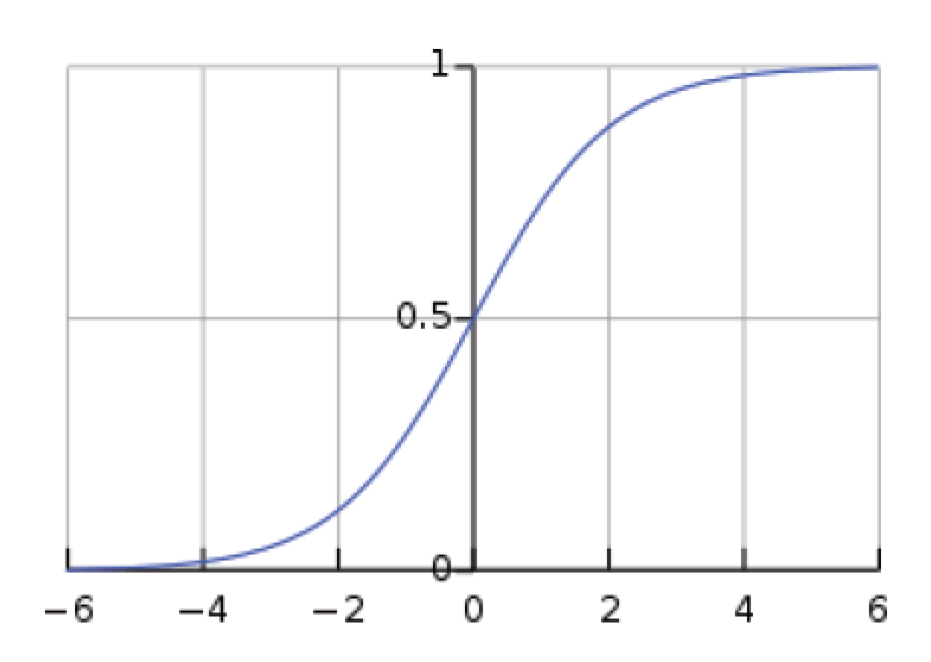
$$\boldsymbol{\omega} = \begin{bmatrix} a_1 \\ a_2 \\ \cdots \\ a_m \\ b \end{bmatrix}, \quad \boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ \cdots \\ x_m \\ 1 \end{bmatrix}$$

注意，向量通常是竖着写的，这里用了转置$\boldsymbol{\omega}^{\mathrm{T}}$将其变成横向，读者或许有点懵，简单讲一下。首先，这里为了简化，向量 $\boldsymbol{\omega}$ 里面多了一位 b，x 里面多了一位 1，这样就可以把常数项放进来，使公式更加简洁。其次，这里使用了线性代数中矩阵的乘法，由于我们的结果 $\boldsymbol{y}$ 只有一个数字，不妨认为它就是一个 1×1 的矩阵。根据矩阵乘法的原则，两个矩阵相乘，前面一个的行数定义结果有多少行，后面一个的列数定义结果就有多少列，所以 m+1 行 1 列的 $\boldsymbol{\omega}$ 要变成 1 行 m+1 列的形式$\boldsymbol{\omega}^{\mathrm{T}}$，这样结果才会是 1×1 的结果。

逻辑回归做的就是对这个结果进一步变化：

$$\hat{y}=\mathrm{sigmoid}(\boldsymbol{\omega}^{\mathrm{T}}\boldsymbol{x})=\frac{1}{1+e^{-\boldsymbol{\omega}^{\mathrm{T}}\boldsymbol{x}}}$$

我们关注 $f(x)=\mathrm{sigmoid}(x)$ 的形式：



其实对于绝大多数的输入 x（绝对值大于 5），sigmoid 函数都会返回一个非常接近 0 或者 1 的结果，只有 x 取值范围在 [-5, 5] 之间时才会是其他值。正是由于这种特性，我们可以近似认为 sigmoid 函数的返回值就是 0 或者 1，这种离散化就是逻辑回归"**名曰回归，实则分类**"的原因——我们就可以认为，这里的 y 代表了某个样本是否属于某个分类的概率，例如鸢尾花分类问题，对一个样本，我们可以对四个特征先进行线性回归，算出一个数，再进一步进行逻辑回归，判断这个样本属于某一分类的概率。注意，这里真实的 y 是一个非 0 即 1 的数字。我们预测的结果 $\hat{\boldsymbol{y}}$ 则是逻辑回归函数的纵坐标——取值范围 [0,1] 的一个概率值。

此时，读者心里可能会有一个疑问：逻辑回归函数中$\mathrm{sigmoid}(\boldsymbol{\omega}^{\mathrm{T}}\boldsymbol{x})$的 $\boldsymbol{\omega}$ 是怎么算出来的？可以猜吗？这里确实需要猜一下，其中的算法大概是这样的：

（1）随机初始化一组 $\boldsymbol{\omega}$，比如可以全部设为 0，当然实际上不推荐这样。

（2）在训练集中，向逻辑回归函数里输入特征 $\boldsymbol{x}$，计算$\boldsymbol{\omega}^{\mathrm{T}}\boldsymbol{x}$，得到预测结果$\hat{\boldsymbol{y}}$。

（3）计算全部训练集中逻辑回归的结果$\hat{\boldsymbol{y}}$和实际 $\boldsymbol{y}$ 的差别。

（4）根据上一步的差别更新 $\boldsymbol{\omega}$。

（5）重复（2）~（4）若干次（iterations）。

2. 损失函数——如何表示预测 $\hat{y}$ 和实际 y 之间的差别

回顾一下我们现在会做什么。首先，猜一组数字，当然会做（当然第 5 章讲述卷积层时还会介绍深度神经网络中随机初始化需要注意的事项）。其次，乘法、加法和 sigmoid 函数也会写，

关键是第三步和第四步，如何表示预测和实际 y 之间的差别，逻辑回归这里引入了交叉熵的概念。

关于交叉熵的相关概念，首先这里的熵指的是信息熵。信息熵是对不确定性的衡量，具体而言，这里引用吴军老师《数学之美》一书中的一段描述：

当我错过了上一届世界杯的比赛，而想知道谁夺得冠军时，我询问一个知道比赛结果的观众。但是他并不愿意直接告诉我，而是让我猜测，每猜一次他要收费 1 元来告诉我，我的猜测是否正确。那么我要花多少钱才能知道谁是冠军呢？

我可以把球队编号，1 到 32 号（当然大家都知道世界杯是 32 支球队，然而过几年变成 48 支的时候我会回来修改的）然后我提问："是在 1 到 16 号中吗？"。如果他告诉我猜对了，我会继续问："是在 1 到 8 号中吗？"。这种询问方式大家都懂，因此这样询问下去，只需要 5 次，也就是只需要 5 元钱就可以知道哪支球队是冠军。

因此，世界杯冠军这条消息的信息量可以看作是 5 元钱。我们回到数学上的问题，使用比特来代替钱的概念（计算机中，一个比特是一位二进制数，一个字节就是 8 个比特），这条信息的信息量是 5 比特。如果有 64 支队伍，就要多猜一次，也就是 6 比特。

可见这里参赛球队越多，冠军归属这条消息的信息熵就越高。同时也要注意，这里信息熵的求法，背后的假设是各球队夺冠的概率相等，而实际情况并非如此，虽然 64 支队伍参赛，但具备夺冠实力的球队只是个位数，因此实际的信息熵是低于这个数字的。

解释完信息熵，我们再来谈一谈什么是交叉熵。交叉熵用来衡量两个正函数是否相似，对于两个完全相同的函数，它们的交叉熵等于零。如果是考虑分类问题中预测的概率 $\hat{y}$ 和实际概率 y 之间的差别，当预测的概率与实际情况一致时，交叉熵即为 0。具体对某个样本而言，定义其交叉熵为：

$$H = -\sum_{k}^{K} y_k \, log \, \hat{y}_k$$

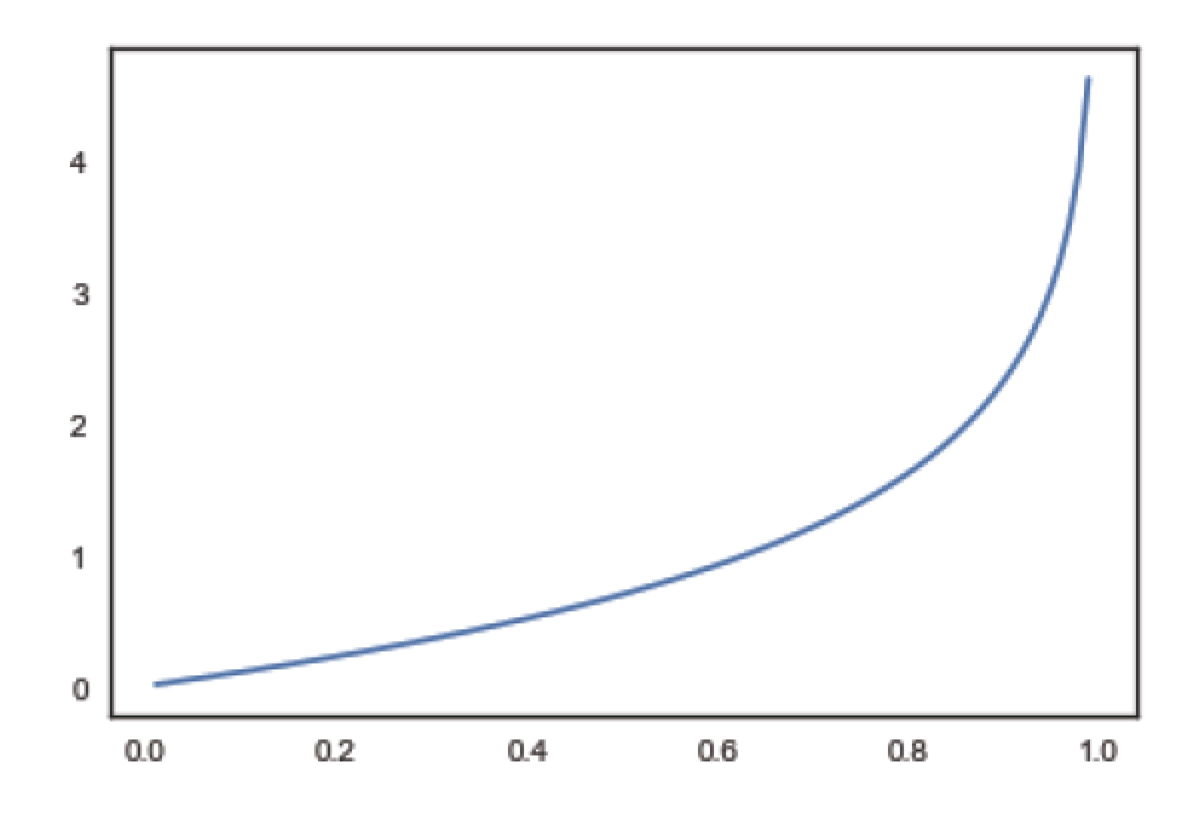
这里的 K 代表 K 个最终分类，如鸢尾花数据集中最终对三种花分类，则 K=3。

如果是二分类的情形，此时 K=2，就有：

$$H = -[y \, log \, \hat{y} + (1-y) \, log(1-\hat{y})]$$

我们简单地预览一下二分类情况下交叉熵的性质，如果真实的分类结果是 y=0，我们看看不同的 $\hat{y}$ 下交叉熵的取值：

```
y = 0
np_yhat = np.linspace(0, 1, 101)
np_h = -(y*np.log(np_yhat) + (1-y)*np.log(1-np_yhat))
plt.plot(np_yhat, np_h)
```



可见我们预测的 $\hat{y}$ 和真实的 y 越接近，即预测得越准确，交叉熵将会越接近 0。相反，如果预测值 $\hat{y}$ 和真实的 y 完全相反，实际不是这种分类情况（$y=0$），预测时却 100% 判断是 1（$\hat{y}=1$），交叉熵的值就会接近正无穷大。

以上内容只是考虑一个样本的情况。当考虑多个样本时，每个样本都可以计算一个交叉熵，此时就可以用全部样本的平均交叉熵作为**损失函数**（Loss function）来衡量模型预测的准确程度：

$$\text{Loss} = \frac{1}{N}\sum_{i=1}^{N} H_i = -\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K} y_{ik}\log\hat{y}_{ik} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K} y_{ik}\log\left(\frac{1}{1+e^{-\omega_k^T x_i}}\right)$$

简单回忆一下，最右边的式子中总共有 N 个样本 K 个分类，在鸢尾花数据集中，就是 N=150、K=3。y_{ik} 是第 i 个样本第 k 个分类，如果这个样本属于第 0 个分类，那么 $y_{i,k}$=0=1，$y_{i,k}$=1,2=0。ω 一共有 K 组，每组都是长度为 M+1（M 是特征数，如鸢尾花数据集 M=4）的向量。在实际计算中，ω 真正需要 $K-1$ 组即可，比如二分类的话，预测一类后另一类的概率也就知道了，由于只预测一类，因此使用一组 ω 即可。

可能读者被这里的数学公式给吓到了，再用通俗一点的语言简单进行总结。其实这里需要交叉熵回答的问题是对于每个样本的分类结果，模型给出的预测与实际情况有多大的区别。所以我们可以将交叉熵理解成机器学习模型的 KPI、GPA 这样的可量化考勤指标，模型每预测一次，就用交叉熵来考勤一次。如果模型的考勤结果不好怎么办？如何改进下一步的工作？请看下一部分内容。

3. 如何根据预测 $\hat{y}$ 和实际 y 之间的差别更新参数 ω

此时，我们就得到了第三步中“计算全部训练集中，逻辑回归的结果 $\hat{y}$ 和实际 y 差别”。我们可以基于这种差别，进一步更新 ω 的值：

$$\boldsymbol{\omega}_{t+1} = \boldsymbol{\omega}_t + \alpha\frac{\partial\text{Loss}}{\partial\boldsymbol{\omega}_t}$$

其中，t 代表迭代次数，因为这里更新值并不是一步完成的，可能迭代了上百次；α 是学习率，用来控制迭代的步长，这里需要根据数据进行一定调整，过小会导致训练缓慢，过大则会造成结果精度不足。

剩下的问题就是求偏导数了。这里引入链式求导法则。为了简化操作，这里只考虑 K=2，即二分类情况，有：

$$\text{Loss} = \frac{1}{N}\sum_{i=1}^{N}\left[y_i \log\left(\text{sigmoid}(\boldsymbol{\omega}^{\mathrm{T}}x_i)\right) + (1-y_i)\log\left(1-\text{sigmoid}(\boldsymbol{\omega}^{\mathrm{T}}x_i)\right)\right]$$

进一步化简，将 y_i 进行向量化、将 x_i 进行矩阵化以消除求和项：

$$\text{Loss} = \frac{1}{N}\left[y \log\left(\text{sigmoid}(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{\omega})\right) + (1-y)\log\left(1-\text{sigmoid}(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{\omega})\right)\right]$$

进行简单的化简，令：

$$g(x) = \frac{1}{1+e^{-x}}$$

利用导数的定义以及链式求导法则，则有：

$$\frac{\partial \text{Loss}}{\partial \boldsymbol{\omega}} = \left(y\frac{1}{g(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{\omega})} - (1-y)\frac{1}{1-g(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{\omega})}\right)\frac{\partial}{\partial \boldsymbol{\omega}}g(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{\omega})$$

根据：

$$\frac{\partial}{\partial x}g(x) = g(x)(1-g(x))$$

带回公式，化简得到：

$$\frac{\partial \text{Loss}}{\partial \boldsymbol{\omega}} = (y - g(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{\omega}))\boldsymbol{X}$$

这里继续来拯救被公式吓懵的读者。前面说到交叉熵损失函数实际就是 KPI、GPA 这样的量化考勤指标，如果想提高得分，应该怎么办？最简单的方法就是看看指标中哪一项得分低——比如考试分数，数学考了 99 分、英语 60 分，我们就可以认为数学成绩相比 100 分满分的差距（$\frac{\partial \text{Loss}}{\partial \boldsymbol{\omega}}$）是 1、英语是 40，那么想提高考试成绩的话，下一阶段就需要将主要的时间精力放在英语上。我们也不能在下一阶段学英语学得太猛而影响总体成绩，所以需要乘以一个学习率 α。

4. 几点思考

至此，我们完成了逻辑回归的主要理论部分。如果读者没有被公式绕晕、坚持到了这里，就从逻辑回归的理论出发，简单谈谈其他的监督学习公式。首先回顾一下逻辑回归的步骤：

（1）随机初始化一组 $\boldsymbol{\omega}$，比如可以全设为 0，当然实际上不推荐这样。

（2）训练集中，在逻辑回归函数里输入特征 x，计算$\boldsymbol{\omega}^{\mathrm{T}}\boldsymbol{x}$，得到预测结果 $\hat{\boldsymbol{y}}$。

（3）计算全部训练集中，逻辑回归的结果 $\hat{\boldsymbol{y}}$ 和实际 $\boldsymbol{y}$ 差别。

（4）根据上一步的差别更新 $\boldsymbol{\omega}$。

（5）重复（2）~（4）若干次（iterations）。

现在的问题是，如果要改动这几个步骤变成一个新算法，可以怎么改？我们注意到，实际上可以改动的地方主要是第二、三步，也就是说，可以有这些思路：

- 预测结果时，把逻辑回归 sigmoid 函数换成一个其他的函数。
- 计算损失时，换成一个损失函数。

这两个步骤通常是同时变换的，在算法层面二者共同推导得到。如支持向量机，就是把逻辑回归的 sigmoid 函数换成核函数，损失函数由平均交叉熵换成了不同分类的距离间隔。又如朴素贝叶斯，预测结果基于概率判断，损失函数同样基于概率判断。本书不再重点讨论这部分内容。

本书后面内容将详细介绍的深度学习算法则基本沿袭了逻辑回归的思路，只改了步骤（2），将原本一个逻辑回归函数变成几十个函数的嵌套，然后利用链式求导法则对嵌套的几十个函数进行反向求导，得出损失函数。然后对其他步骤做了一些工程创新，使其可以适应更大规模数据。

5. 正则化

前面提到，过多的参数会导致过拟合，因此可以在规定损失函数的时候，将这一点考虑进来：

Loss = 正则化系数 (C) × 分类准确率罚分项 + 过多参数罚分项

在上一步中其实也注意到，逻辑回归随机了一个初始化的系数 ω 后，接着借助求导进行梯度下降，就可以得到最终解，需要的额外参数只有 α。而实际工程运用中，借助 libfgs sag 等梯度下降求解工具，我们甚至也不需要提供 α 值。因此似乎逻辑回归不需要提供额外的超参数。实际上，如果考虑正则化，就需要关注正则化系数 C 的影响。因此，**逻辑回归调参主要调的就是这里的 C 值**，过小的 C 会过度强调罚分项的损失而非对模型预测结果的关注，造成欠拟合，而过大的 C 则会过分强调结果，可能会造成参数数目过多，进而造成过拟合。

这里的过多参数罚分项有两种比较常见。

一种是各项系数的绝对值相加，即 L1 正则化：

$$\text{Loss} = ||\boldsymbol{\omega}||_1 - \frac{C}{N}\left[y\log\left(\text{sigmoid}(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{\omega})\right) + (1-y)\log\left(1-\text{sigmoid}(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{\omega})\right)\right]$$

另一种是使用 L2 正则化：

$$\text{Loss} = \frac{\boldsymbol{\omega}^{\mathrm{T}}\boldsymbol{\omega}}{2} - \frac{C}{N}\left[y\log\left(\text{sigmoid}(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{\omega})\right) + (1-y)\log\left(1-\text{sigmoid}(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{\omega})\right)\right]$$

此时，求导的话，导数将会分别变成：

$$\frac{\partial \text{Loss}}{\partial \boldsymbol{\omega}} = C(y - g(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{\omega}))\boldsymbol{X} + sgn(\boldsymbol{\omega})$$

$$\frac{\partial \text{Loss}}{\partial \boldsymbol{\omega}} = C(y - g(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{\omega}))\boldsymbol{X} + \boldsymbol{\omega}$$

这里 $\text{sgn}(\boldsymbol{\omega})$ 代表 $\boldsymbol{\omega}$ 正负号。

继续拯救被公式绕晕的读者们，我们知道 KPI、GPA 的考核指标虽然是越多越全面越好，但如果搞出几百项考核指标来，首先这些指标的内容就不好理解，让人无法根据结果得分做下一阶段的规划；其次这些考核指标是否全部合理也是问题。因此考核指标需要简化，机器学习中就使用了正则化策略来简化考核指标的复杂性。最后，再解释一下 L1 正则化与 L2 正则化的区别。相比 L2 正则化，加入 L1 正则化后，优化得到的 ω 向量，会具有更高的稀疏性，即向量的很多参数会是 0。而 L2 正则化后，优化得到的向量参数则会是一个接近 0 的、很小的参数。具体原因是在 L2 正则化中，损失函数对 ω 的偏导数会随着 ω 的值减小而不断减小，梯度下降速度越来越慢，因此最后结果接近但不等于 0。而 L1 正则化的梯度，则只和 ω 的正负有关，与其本身值大小无关，因此梯度下降速度始终会保持一个最小值，保证最终结果的稀疏性。

本书在 5.4 节会结合深度神经网络，再次讨论利用正则化、防止过拟合的问题，请读者留意。

2.3.4 第四步，代码实战

本节将根据之前讲述的部分造一个轮子以方便大家理解，然后给出 sklearn 的代码，用于实战时使用。

```
C = 1             # 正则化系数 C
alpha = 0.1       # 学习率，控制更新步长
y = data.target
y[y==2] = 1       # 简化问题，只考虑是否是 setosa

# 矩阵化特征。加一列全部是 1 的特征，化 ax+b 为 w(x+1)，简化表达
X = np.hstack([data.data, np.ones_like(y).reshape(len(y),1)])

# 第一步， omega 随机初始化
np.random.seed(42)
omega = np.random.random(X.shape[1]).reshape(5, 1)

for i in range(10):
    # 第二步，在训练集中，逻辑回归函数里，输入特征 x，计算 sigmoid(omega^Tx)，得到
预测结果
    y_hat = 1 / (1+np.exp(-X.dot(omega)))

    # 第三步，计算全部训练集中逻辑回归预测的结果和实际 y 的差别，使用 L2 正则化
    dL = X.T.dot(C * (y.reshape(-1,1) - y_hat))+ omega

    # 第四步，根据上一步的差别更新 omega
    omega += alpha * dL

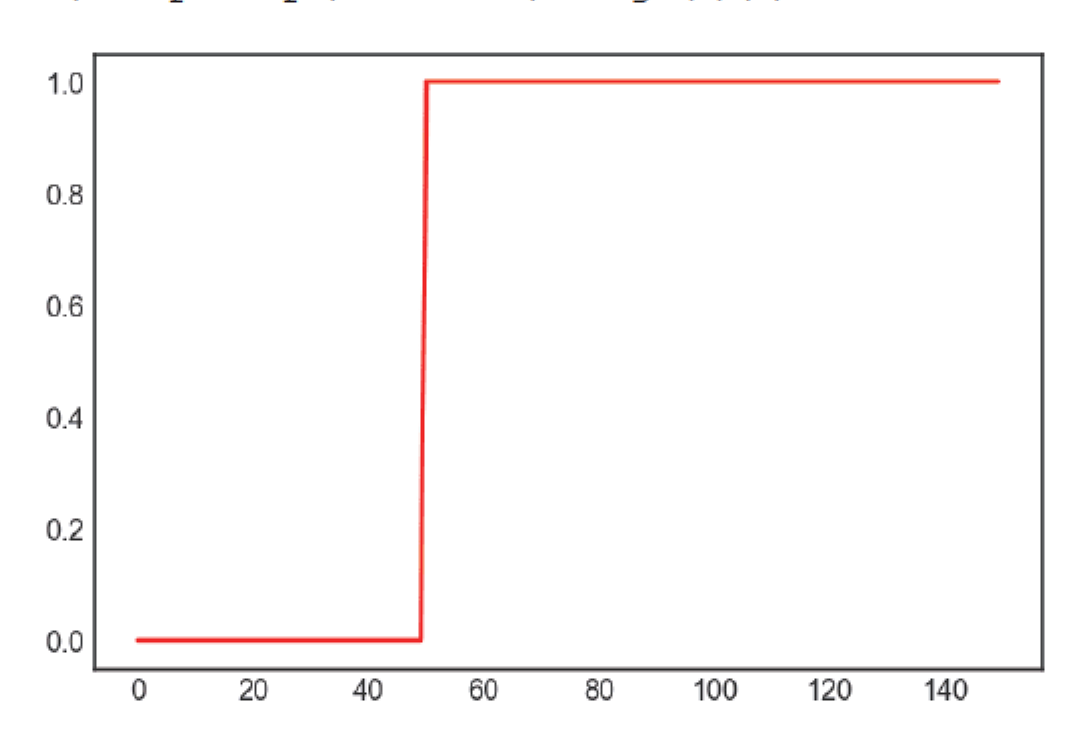
omega
```

运行结果：

```
# out:
array([[-16.8213461 ],
[-39.77445862],
[ 62.60998404],
[ 30.07176958],
[ -8.52757522]])
```

预览结果：

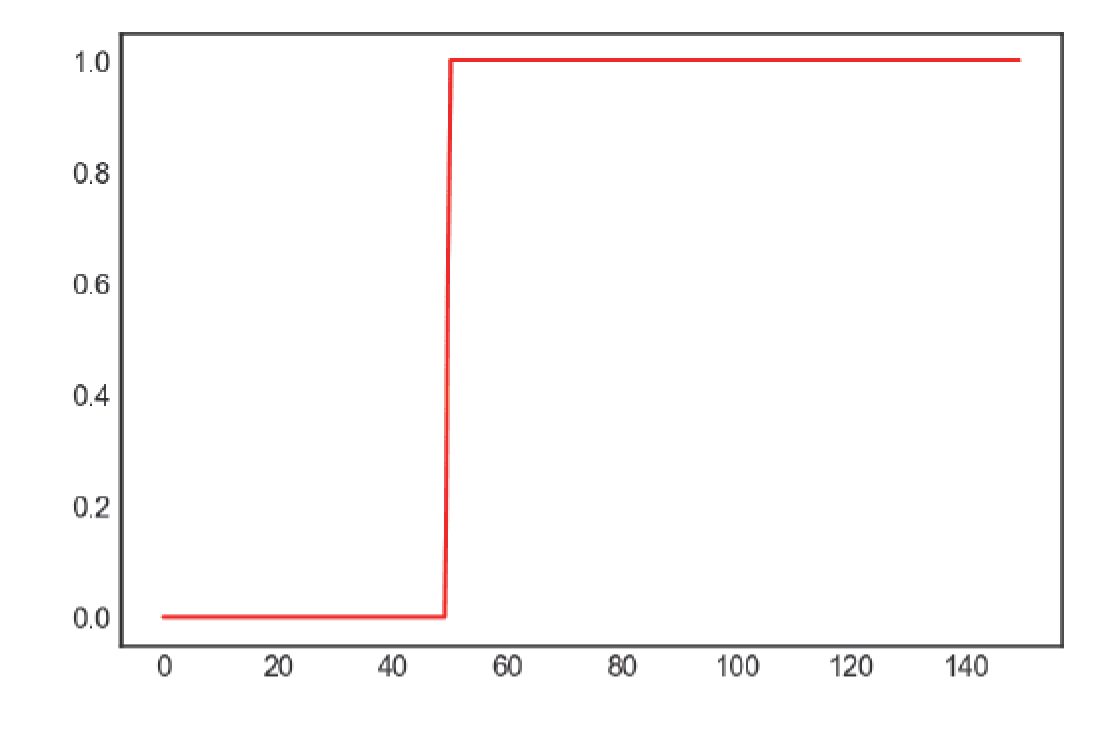
```
plt.plot( 1 / (1+np.exp(-X.dot(omega))))
```



结果中前 50 个样本被预测为分类 0，即 setosa，其他的被预测为非 setosa，与预期相同，造轮子完成。

实战使用时，我们可以直接调用 sklearn 的相关包，发现结果同样准确：

```
from sklearn.linear_model import LogisticRegression
model = LogisticRegression(C=1)
model.fit(X, y)
plt.plot(model.predict(X))
```



1. K 折交叉与网格搜索

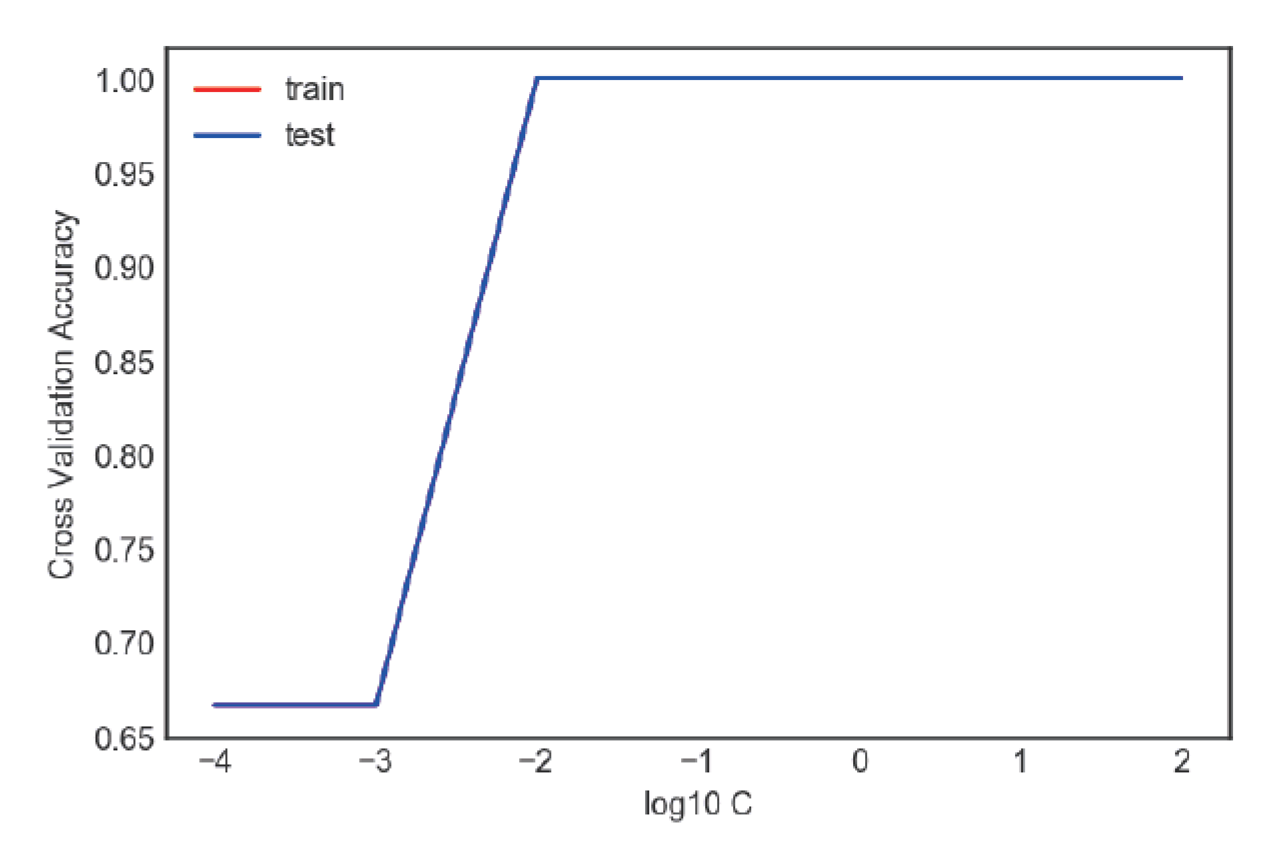
之前的代码其实存在一些问题，我们前面强调过，但是在代码中并没有体现，大家应该也注意到了：

（1）训练集和测试集要隔离，而上一部分代码测试用的数据完全就是训练集，这样很容易过拟合。

（2）正则化系数是相当重要的参数，这里直接用 C=1 是否合理？

因此，为了解决问题 1，我们引入 K 折交叉，将数据平均分成 K 份，K-1 份拿来训练，1 份看结果，然后重复 K 次，用这种方法实现训练集和测试集的隔离；为了解决问题（2），我们引入参数的网格搜索（Grid Search），尝试不同参数的选择，寻找最优的一种。

```
from sklearn.model_selection import GridSearchCV
parameters = {'C':[0.0001, 0.001, 0.01, 0.1, 1, 10, 100]}
model = LogisticRegression()
clf = GridSearchCV(model, parameters, cv=10)
clf.fit(X, y)
plt.plot(
   np.log10(np.array([0.0001, 0.001, 0.01, 0.1, 1, 10, 100])),
   clf.cv_results_['mean_train_score'],
   label="train"
)
plt.plot(
   np.log10(np.array([0.0001, 0.001, 0.01, 0.1, 1, 10, 100])),
   clf.cv_results_['mean_test_score'],
   label="test"
)
plt.xlabel("log10 C")
plt.legend()
plt.ylabel("Cross Validation Accuracy")
```



由此可见，这里不存在过拟合问题，10 折交叉验证自动生成的训练集、验证集预测准确率高度重合。C=0.0001, 0.001 时会由于 C 过小、对预测结果的关注不足造成欠拟合问题。我们之前用的 C=1 其实是碰巧用了合适的参数。

更多内容，读者可以参考作者博客文章 https://zhuanlan.zhihu.com/p/25637642。

2. 评价不平衡样本分类结果

之前根据准确率优化模型，如果数据分布不平均，单纯地使用错误率作为标准会产生很大的问题。比如某一种罕见疾病的发病率是万分之一（0.01%），这时如果一个模型什么都不管，直接认为这个人没有病，也能拿到一个 99.99% 正确的模型。这种情况下，模型预测准确率是很高，但却没有什么实际价值。那么，应该如何正确衡量准确率的这个问题呢？如果是经典的二分类问题，这种情况下需要综合考虑不同标准下的**灵敏度**以及**假阳性率**。

理解这两个概念之前，我们要明确一点，就是与医生直接判断某一患者是否有病相比，模型给出的是否有病推论，是一个概率，这时可以选择一个判断有病的阈值，比如 1%，即 1% 可能有病的情况下即判断为有病，和真实情况对比后得到如下的二联表：

	预测有病（+）	预测无病（-）
真实有病	TP	FN
真实无病	FP	TN

在此基础上有：

真阳性率 TPR：在所有实际为阳性的样本中，被正确地判断为阳性的比率。

$$\mathrm{TPR} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}}$$

假阳性率 FPR：在所有实际为阴性的样本中，被错误地判断为阳性的比率。

$$\mathrm{FPR} = \frac{\mathrm{FP}}{\mathrm{FP} + \mathrm{TN}}$$

然后在从 0% 到 100% 区间取多个阈值，得到不同的 FPR/TPR 值，以 FPR 为 x 轴、TPR 为 y 轴，就可以得到一条 ROC 曲线，继而计算 ROC 围成的面积—— AUC 值。这种情况下可以认为，AUC 值越高越好，最高可以是 1。如果 AUC 值接近 0.5，则分类器等于随机猜测。遇到特别的情况，如果 AUC 接近 0，则很可能预测与真实情况完全反了，当然这种情况基本很少出现。

注意这里 AUC 越高越好和最小化损失函数这两个概念。可能某个模型的损失函数已经最小化了，但是 AUC 却不高，造成模型在实际使用时会引入很多错误—— 一个较低的 AUC 值，可能会在追求高灵敏度时引入大量的假阳性，其后果就是医生通知了十个患者有患癌风险，最终可能只有一个真有问题，其他九人虚惊一场。这种情况就是模型并未被很好地训练，可能存在着欠拟合的问题。同时，也可能损失函数最小化以后，测试数据 AUC 也很高，但是实际运用在真实案例中却又有大量的错误，就是所谓的过拟合了。

如果损失函数无法满足我们的评价标准（高 AUC 值），此时需要做的一件事就是调整损失函数，比如给数量较少类别的样本赋予更高的权重等。那么为什么不直接用 AUC 值作为优化目标呢？原因很简单，损失函数和整个模型的结构是偶联的，AUC 值虽然作为评价指标很好，但是计算步骤相对要麻烦很多，也不利于向模型中传播误差梯度，所以一般不直接优化 AUC 值，而是用均方误差（MSE）、交叉熵（Cross Entropy）等更容易计算的指标进行优化。

使用 sklearn 计算阈值的方法：

```
    from sklearn.metrics import roc_curve,auc
    np.random.seed(42)
    # 假如真实情况 10000 个病人，有 10 个是有病的
    np_real = np.array([0.0  for i in range(9990)] + [1.0 for i in range(10) ],
dtype=bool)

    # 预测 1，  全预测为没问题，准确率 99.90%
    np_pred_allf = 0.1*np.random.random(10000)

    # 预测 2:，准确预测使用情况，准确率 100%
    np_pred_true = np_pred_allf.copy()
    np_pred_true[-10:] = 0.99

    fpr, tpr, thresholds = roc_curve(np.array(np_real, dtype=int),np_pred_
allf )
    AUC_value = auc(fpr, tpr)

    fpr2, tpr2, thresholds2 = roc_curve(np.array(np_real, dtype=int),np_
pred_true )
    AUC_value2 = auc(fpr2, tpr2)

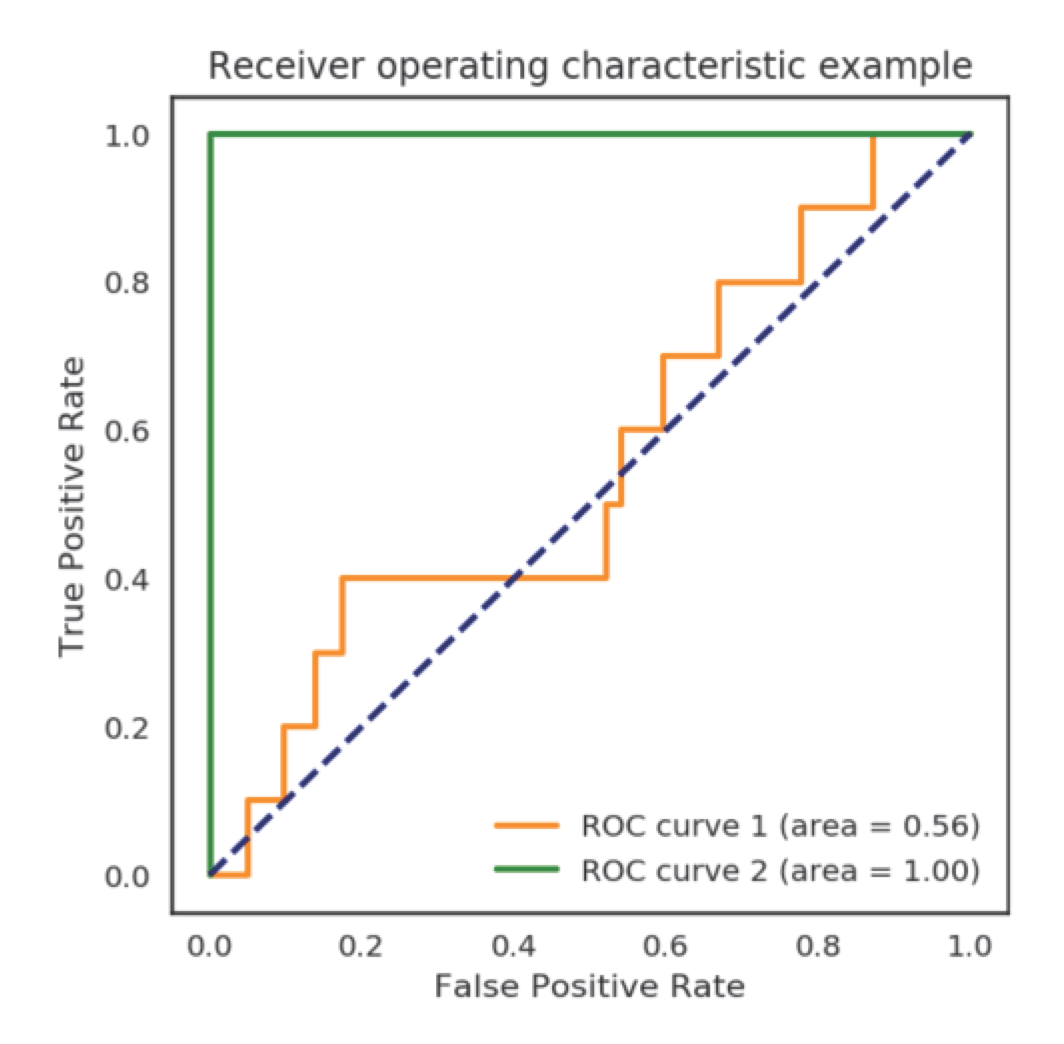
    # 虽然准确率差不多，但是 AUC 值差异巨大
    print(AUC_value, AUC_value2)
    # 0.555805805806 1.0

    fpr2 = np.array([0] + list(fpr2))
    tpr2 = np.array([0] + list(tpr2))

    plt.figure()
    lw = 2
    plt.plot(fpr, tpr, color='darkorange',
             lw=lw, label='ROC curve 1 (area = %0.2f)' % AUC_value)
    plt.plot(fpr2, tpr2, color='g',
             lw=lw, label='ROC curve 2 (area = %0.2f)' % AUC_value2)

    plt.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--')

    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('Receiver operating characteristic example')
    plt.legend(loc="lower right")
    plt.show()
```



2.4 参考文献及网页链接

［1］ 2.2. Manifold learning¶. 2.2. Manifold learning — scikit-learn 0.19.0 documentation. Available at: http://scikit-learn.org/stable/modules/manifold.html.

［2］ Cross entropy. Wikipedia (2017). Available at: https://en.wikipedia.org/wiki/Cross_entropy.

［3］ Foundation, N. I. P. S. NIPS 2017. Available at: https://nips.cc/Conferences/2016/Schedule?showEvent=6203.

［4］ Logistic regression. Wikipedia (2017). Available at: https://en.wikipedia.org/wiki/ Logistic_regression.

［5］ One Hot Encoding in Scikit-Learn. ritchieng.github.io. Available at: http://www.ritchieng.com/ machinelearning-one-hot-encoding/.

［6］ Training, test and validation sets. Wikipedia (2017). Available at: http://en.wikipedia.org/wiki/Training,_test_and_validation_sets.

［7］ Underfitting vs. Overfitting. Underfitting vs. Overfitting — scikit-learn 0.19.0 documentation. Available at: http://scikit-learn.org/stable/auto_examples/model_selection/plot_underfitting_overfitting.html.

［8］吴军 . 数据之美 [M]. 2 版 . 北京：人民邮电出版社 , 2014.

［9］周志华 . 机器学习 [M]. 北京：清华大学出版社，2016.

［10］机器学习基础——PCA（主成分分析）. g11d111 的博客——CSDN 博客 . Available at: http://blog.csdn.net/g11d111/article/details/66473643.

第 3 章

数形结合——图像处理基础知识

通过上一章的学习，读者对于机器学习的基本概念有了一定的了解，此时如果拿到一个简单的 M×N 矩阵化数据集（M 个特征，N 个样本），总体流程应该怎么走，心里已经有了一个基本的概念。

不过，如果要分类的样本不是简单的几个数字，如鸢尾花数据集中花瓣、花萼的长宽，而是一张图片（如图 3-1 所示），这时应该怎么办？





（图片来源：https://www.tensorflow.org/get_started/estimator）

图 3-1 分类的样本是图片

这里提供三种解决问题的思路：

（1）手动提取重要的特征，用数字表示。如鸢尾花数据集，当年就是用尺子量出来的长度、宽度，交给机器学习分类器。

（2）用简单的图像处理操作，将图片转换为少数几个简单的轮廓特征，交给机器学习分类器。

（3）用深度神经网络，让深度学习模型自动提取图片的各种特征，再用模型自动提取的特征训练分类器。

我们在上一章主要介绍了第（1）种思路，本章主要讨论第（2）种思路，后面的部分则重点讨论第（3）种基于深度学习的方法。而对图像进行简单处理操作，实际上就是利用计算机程序实现类似 Photoshop 的粗略操作，当然这里用 Photoshop 的目的并非是让图像更加美观，而是要让图像尽可能简单，只保留最重要的特征，方便模型提取特征。

3.1 读取图像文件进行基本操作

首先需要介绍图像文件的存储格式。计算机图像可以分为矢量图和位图。矢量图以代码的形式存储，常见的包括学术论文的统计图、公司图标等。这里主要讨论如何处理位图，其存储格式包括 png、jpg、tiff 等，常见的包括各种手机、数码相机拍摄的相片。而我们通常看见的 gif 动画、mp4 影片，也是由多个位图按照时间排列、以固定的帧数进行播放的。

这里 png、jpg、tiff 格式的彩色照片，本质上都是一个三维的矩阵，即长度、宽度以及颜色（RGB，红绿蓝）。格式的文件结构都是十六进制数，分别使用了不同的图像压缩方法。当然解析这部分内容并不需要大家过多关注，我们可以在 Python 中使用现成的工具，将十六进制数格式的文件转换成三维矩阵。我们重点讲一下 opencv 的用法，这里使用 opencv-python 官网的例子。

3.1.1 使用 python-opencv 读取图片

```
    import numpy as np
    import cv2

    %matplotlib inline
    !wget https://raw.githubusercontent.com/abidrahmank/OpenCV2-Python-
Tutorials/master/data/messi5.jpg
    img = cv2.imread('messi5.jpg')
    print(img.shape)
```

运算结果：

```
#output:
(342, 548, 3)
```

完成图像读取以后，可以用 Python 的 matplotlib 基本绘图库进行图像的展示，如图 3-2 所示。

```
import matplotlib.pyplot as plt
%matplotlib inline

plt.imshow(img)
```

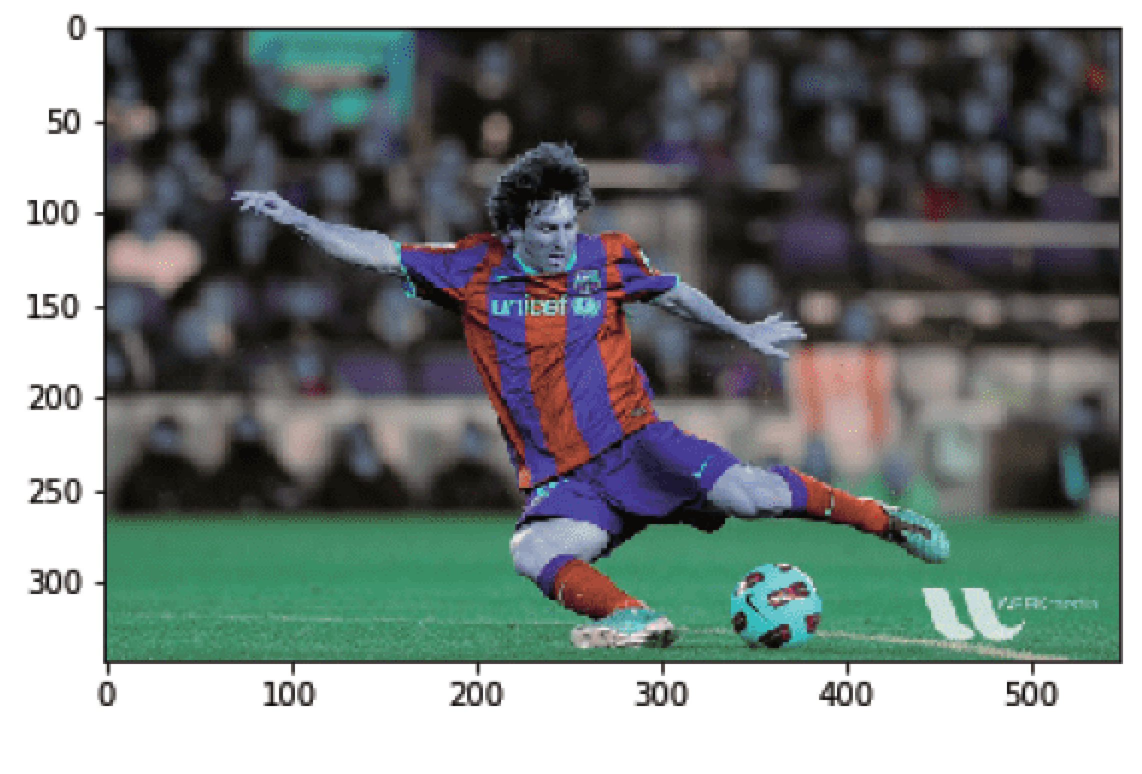


图 3-2 展示图片

读取图片成功，这里的三维矩阵高度为 342 像素，宽度为 548 像素，使用 RGB 编码。如果读者是球迷的话，很容易会发现问题，首先这个球应该是黄色的，怎么是浅蓝色？巴萨球迷更能一眼看出球袜的颜色反了，不是红色是蓝色——红蓝色反了。

出现这个问题的原因是，opencv-python 默认使用 BGR 编码，相对于 RGB 而言，色彩这个维度正好是反的。之所以这里看起来不太对劲的原因，主要还是归功于巴萨球衣是红蓝相间的，反过来仍然是红蓝。如果是米兰两队（红黑与蓝黑）这样转换，球队就会弄错了。

3.1.2 借助 python-opencv 进行不同编码格式的转换

我们可以用简单的矩阵操作，或者直接用 opencv 的函数，将 BGR 颜色编码转换成 RGB，如图 3-3 所示。

```
# 方法 1. 直接操作数组
fig = plt.figure(figsize=(10, 6))
ax1 = fig.add_subplot(121)
ax2 = fig.add_subplot(122)
ax1.imshow(img[:,:,np.array([2,1,0])])
# 方法 2. 调用 opencv 函数
ax2.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
```

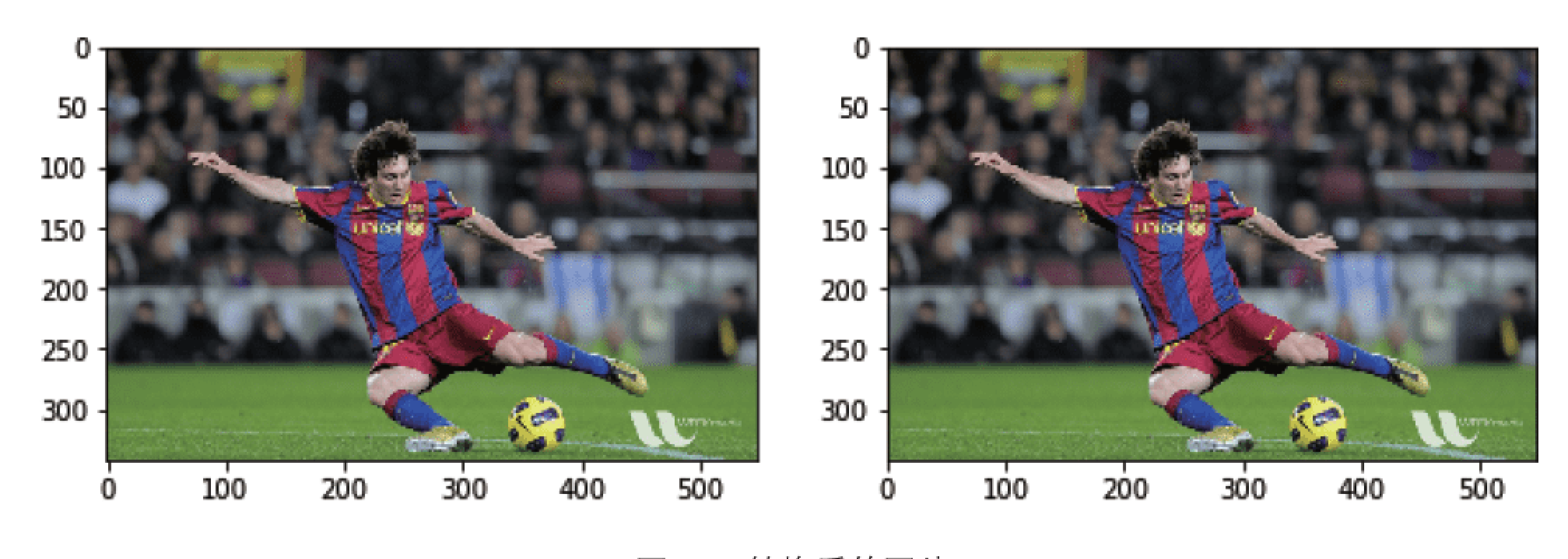


图 3-3 转换后的图片

3.1.3 借助 python-opencv 改变图片尺寸

如果此时想转换成一个小一点的黑白相片，将高度×宽度变成240×360，可以进行如下设置：

```
img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
img_gray_small = cv2.resize(img_gray, (360, 240))
```

存储图片：

```
cv2.imwrite("gray_out.png",img_gray_small)
```

运算结果：

```
# out:
True
```

此时，转换成黑白相片并经过缩小后的文件已经被成功存储了。

3.2 用简单的矩阵操作处理图像

3.2.1 对图像进行复制与粘贴

如果我们想在上面的图片中加入其他元素，应该如何做呢？比如在 3.1 节的图中再加入一个球，顺便添加一些文字。

这里要加入一个球的话，由于图中已经有球了，所以实际上从图中复制即可。我们注意到在这张 342×548 的图中，球的纵坐标在 300 左右、横坐标在 360 左右，大小约为 60 像素，可以用 numpy 的数组切片 img[280:340, 330:390]，得到一个子矩阵，继而给原图中的其他位置赋值。添加文字的话，在查询 opencv 的文档后，可以用 cv2.putText 实现这一功能，如图 3-4 所示。

```
ball = img[280:340, 330:390]
img[273:333, 100:160] = ball
cv2.putText(img," 文字 ",(10,50),cv2.FONT_HERSHEY_PLAIN,
            4,(255,255,255),2,cv2.LINE_AA
)
plt.imshow(img[:,:,np.array([2,1,0])])
```

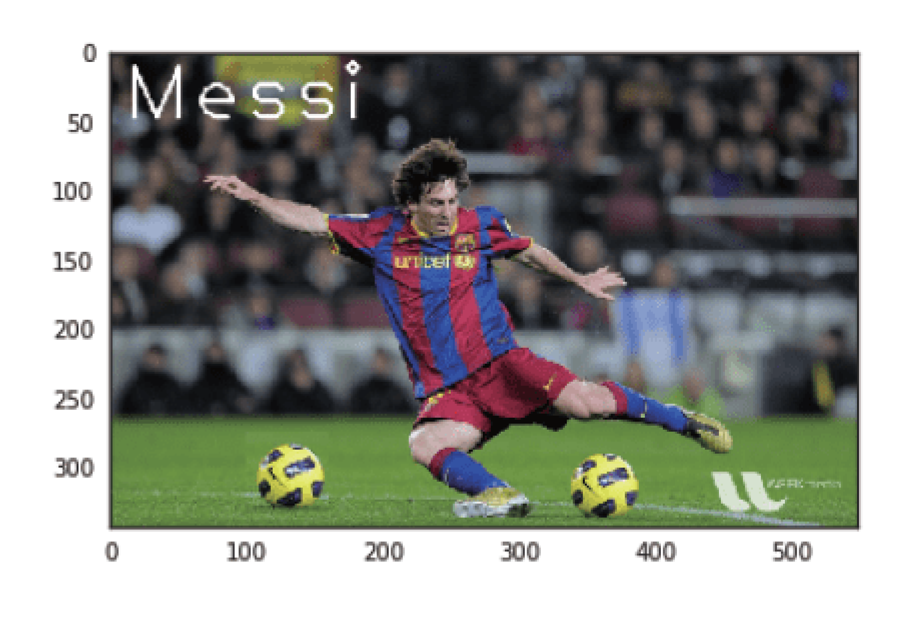


图 3-4　复制足球并添加文字

3.2.2 把图像当成矩阵进行处理——二维码转换成矩阵

3.2.1 小节进行的复制与粘贴只是处理矩阵的最简单操作。本书将讲一些稍微难一点的操作，就是将二维码转换成矩阵。当然，这里只是将二维码转换成 0/1 的矩阵而已，至于变成矩阵后如何从矩阵提取其中的内容，大家感兴趣的话可以进一步研究。



图 3-5 原始大小（220,220）二维码

我们以如图 3-5 所示的景略集智数据科学的官方主页（jizhi.im）二维码为例，这里实际上是由 25×25 个像素点组成的，每个像素点或黑或白，但是这张图片并不止 25×25 个像素点，而是 220×220 个，那么问题就来了，如何将 220×220 个点映射到 25×25 个点的坐标上去？

我们首先读取图片，并转换为 [0,255] 的灰度值编码：

```
img_jizhiQR = cv2.imread("./3.png")
img_jizhiQR = cv2.cvtColor(img_jizhiQR, cv2.COLOR_BGR2GRAY)
print(img_jizhiQR.shape)
```

运算结果：

```
# out:
(220,220)
```

接下来，从 220×220 个点映射到 25×25 个点，实际上直接做一个简单线性变换即可，即如果坐标是（20，20），则映射后的坐标点应该是（20/220*25, 20/220*25） =（0, 0），并且实际上很多点变换后对应（0, 0）这个点。

此步骤的代码如下：

```
# 看看取值情况，发现只有 0（黑色）和 255（白色）
set(img_jizhiQR.ravel())
# {0,  255}

# 规定 220 个坐标，分别对应到 25 个点坐标上
np_220to25_idx = np.linspace(0, 24.9, 220).astype(np.int)

# 对两个维度一起规定，如此 220×220 二维平面上的每个点就都有了到 25×25 的映射
np_220to25_mesh = np.meshgrid(np_220to25_idx,np_220to25_idx)
np_25_counts = np.zeros([25, 25])
for row_idx in range(220):
    for col_idx in range(220):
        col_num_25 = np_220to25_mesh[0][row_idx][col_idx]
        row_num_25 = np_220to25_mesh[1][row_idx][col_idx]
        if img_jizhiQR[row_idx][col_idx] == 255:
            # 如果原先是白色，则这里 25×25 矩阵中对应点 +1
```

```
            np_25_counts[row_num_25][col_num_25] += 1

# 统计 25×25 矩阵中每个点有多少在映射前是白色
import seaborn as sns
sns.set_style('white')
sns.distplot(np_25_counts.ravel())
```

运行结果如图 3-6 所示。

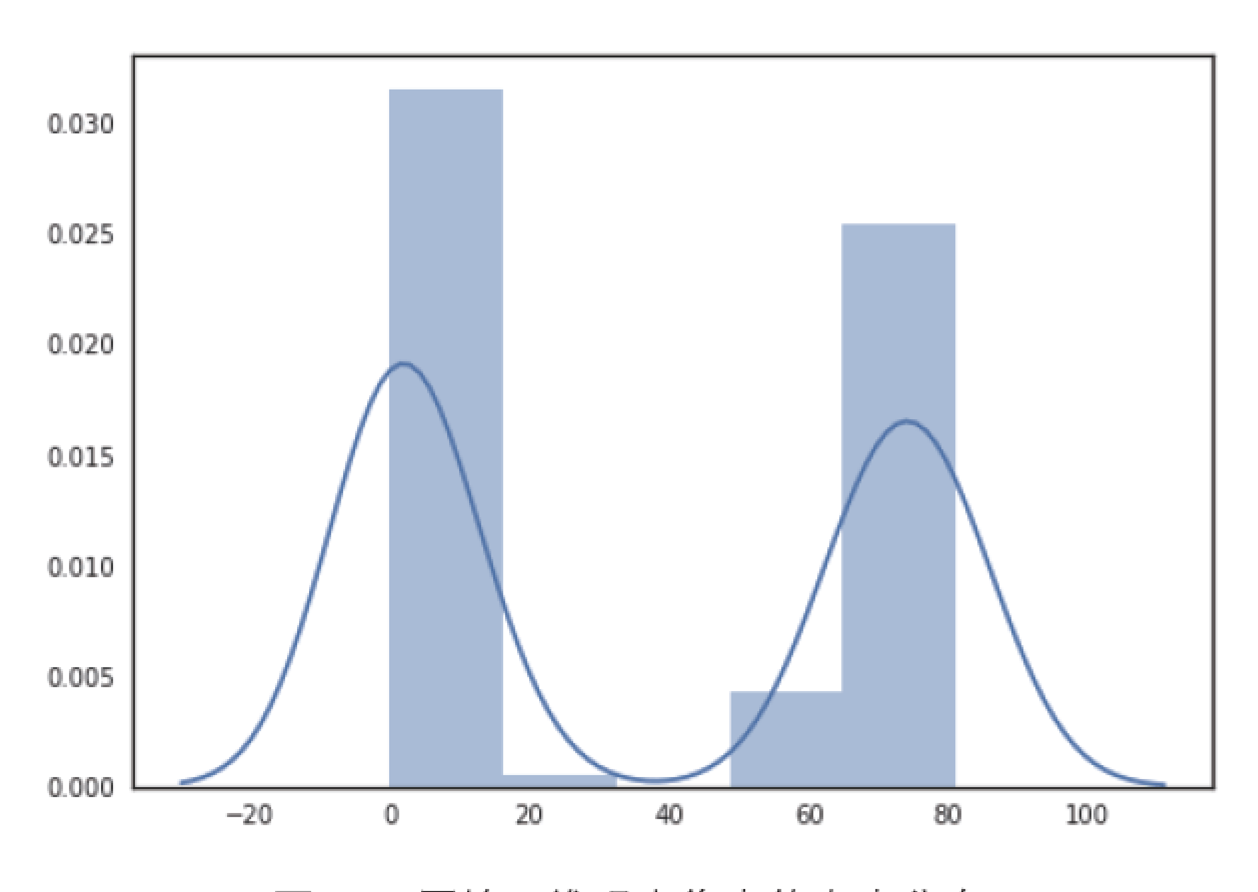


图 3-6　原始二维码中像素值大小分布

这里有一部分点取值小于 40，而另一部分点取值大于 40。统计这一步的原因是，我们的映射会产生一些错位，即切割不准确，此时有一些本来应该是黑色的点，切的时候进来了一些白色，造成干扰。此时需要设定一个阈值，即只有大于若干白色点，我们才认为是白色，否则是黑色。很明显，这里可以设定这一阈值为 40，查看结果如图 3-7 所示。

```
plt.imshow(np_25_counts>40, cmap="gray")
```

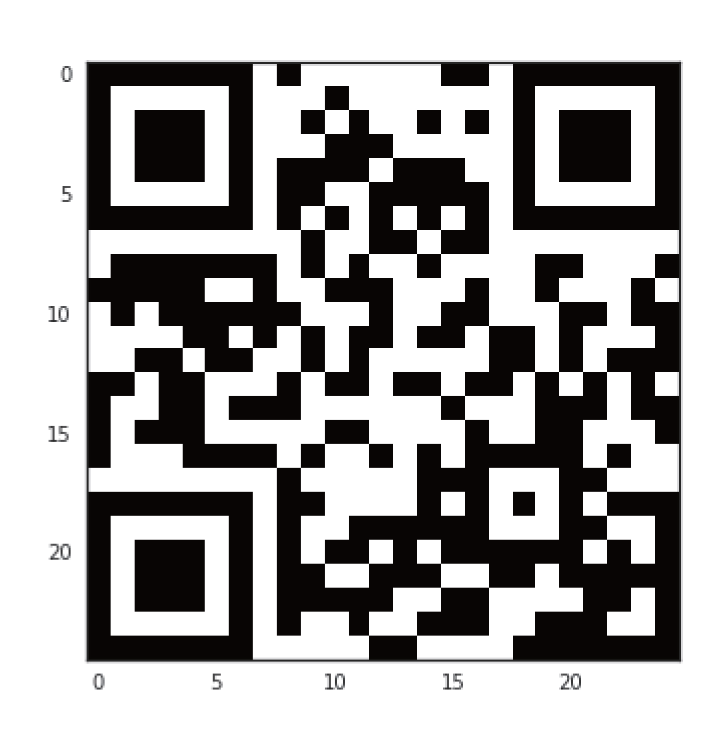


图 3-7　原始二维码压缩到（25×25）的结果（设定二值化阈值为 40）

我们注意到这个结果与变换前的二维码看起来完全相同，扫描后仍然是景略集智数据科学的主页 https://jizhi.im。而实际上，这张图片的形状已经由之前的 220×220 缩小到了 25×25。此时就可以对照二维码的编码标准，用我们缩小后的矩阵逐一解读二维码信息。

这里进一步展开说明：

第一，为什么有很多二维码工具我们还要讲这方面的内容。其实不是要解决生活中二维码的问题，而是以二维码这个最像矩阵的图像为例谈谈如何将图像当成矩阵处理；并且现在越来越多的数据分析公司喜欢把自己的招聘网页设计成奇怪的二维码，然后有人解出来的在招聘时加分，至于如何解决这类问题，其难点不外乎找出映射函数（比如映射为圆形）以及各种阈值（奇怪的颜色形状）。

第二，有 SQL 经验的读者会隐约感觉到，这里实际就是对图片坐标进行映射后进行了类似 AVG(value) GROUP BY KEY 的操作。有深度学习基础的读者，更是能意识到，这个问题实际上就类似深度神经网络中的一个池化（Pooling）过程，会在 5.3 节详细介绍。那么这个问题能不能用深度神经网络框架 Tensorflow 解决呢？答案是可以的，具体代码的含义后续会解释，当然这本书后续案例基本上都是使用 Keras 包装 Tensorflow，以简化难度。

```
import tensorflow as tf

# 这里类似解方程时设置未知数
mat_input = tf.placeholder(tf.float32)

# 根据 tensorflow 文档如此写，即我们用一个大小 9×9、相互不重叠（步长均为 9）的卷积核，
扫描输入。
op = tf.nn.avg_pool(value=mat_input, ksize=[1,9,9,1], strides=[1,9,9,1],
padding='SAME')

# 计算结果
with tf.Session() as sess:
    # 初始化
    sess.run(tf.global_variables_initializer())
    # 将数据转换为 [N(number, 图片数目), H(height,图片高度), W(width,图片宽度),
C（channel，颜色通道数）] 的格式，得到结果
    # 因此需要将二维的图片，在数目、颜色通道两个维度都加上
    res = (sess.run(op, feed_dict={mat_input: img_jizhiQR[np.newaxis,
:,:,np.newaxis]}))
```

查看结果（如图 3-8 所示）：

```
sns.distplot(res.ravel())
```

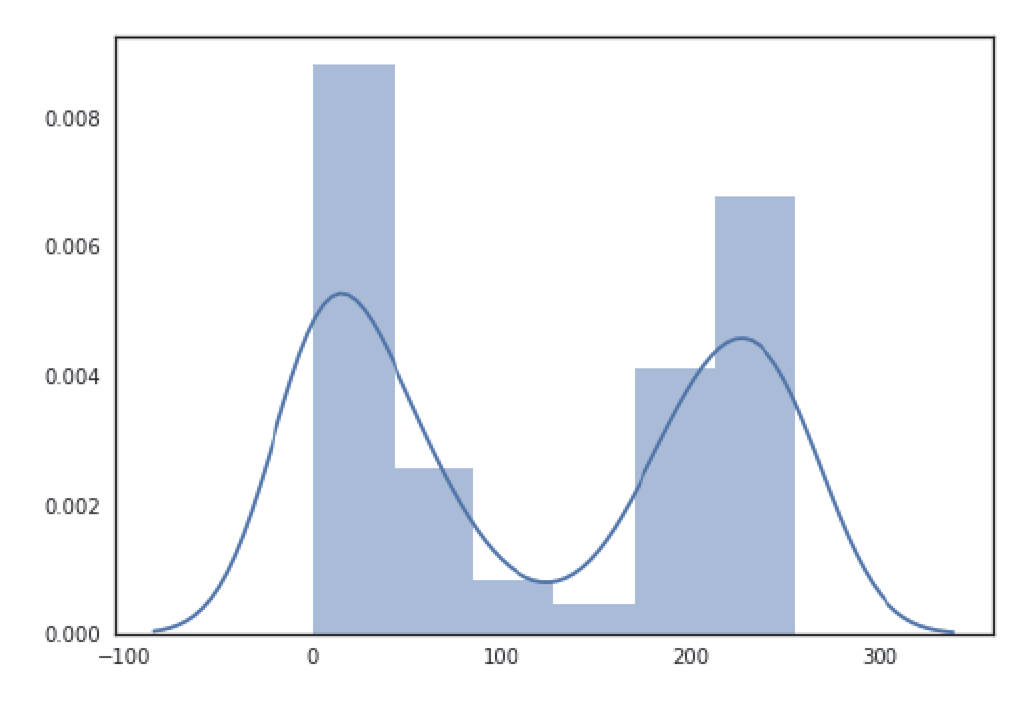


图 3-8 查看结果

这里的阈值目测可能在 130 左右，可以用 kmeans 算法找到这个阈值，然后进行分类，其结果如图 3-9 所示：

```
from sklearn.cluster import KMeans
kmeans = KMeans(n_clusters=2, random_state=0).fit(res.ravel().reshape(-1,1))
min(res.ravel()[kmeans.labels_==1])
minVal = min(res.ravel()[kmeans.labels_==1])
print(minVal)
# 122.77778
plt.imshow(res[0,:,:,0] > 122.78, cmap="gray")
```

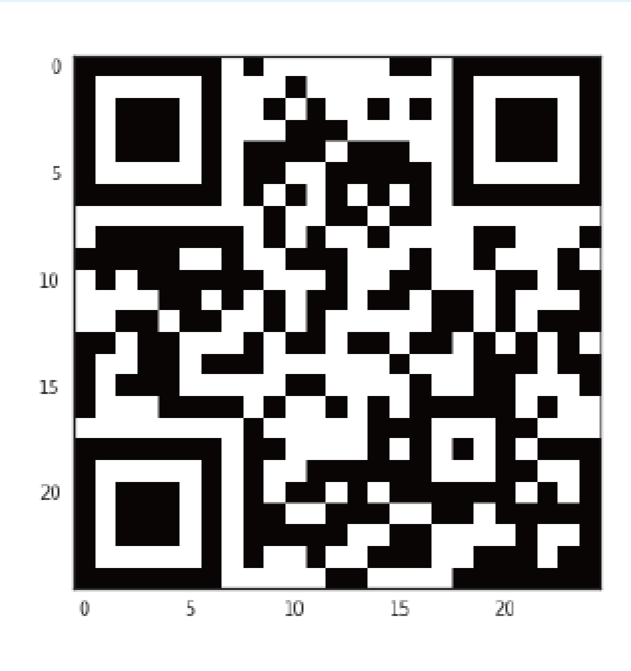


图 3-9　二维码图

此时扫描此图，结果仍然是 https://jizhi.im。

最后，我们增加难度。读者应该注意到，各个网站公众号实际使用的二维码边缘上有空白，中间有 Logo。这种情况下，应该如何排除这些因素？作者这里提供一种思路，首先将之前的程序整理成函数：

```
def AvgPool(img_input, num_out):
    mat_input = tf.placeholder(tf.float32)
    k_size = int(img_input.shape[0]/num_out)+1
    op = tf.nn.avg_pool(value=mat_input, ksize=[1,9,9,1], strides=[1, 9, 
9, 1], padding='SAME')
    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        res = (sess.run(op, feed_dict={mat_input: img_input[np.
newaxis,:,:,np.newaxis]}))

    bigger_label = 1
    kmeans = KMeans(n_clusters=2, random_state=0).fit(res.ravel().
reshape(-1,1))
    if kmeans.cluster_centers_[0] > kmeans.cluster_centers_[1]:
        bigger_label = 0

    threshold = min(res.ravel()[kmeans.labels_==bigger_label])
    return res[0,:,:,0] > threshold
```

这里仍然是使用 TensorFlow 执行卷积计算，然后对结果进行 Kmeans 分类，找出阈值，进行**黑白的二值化**。之前的代码比较分散，因此这里写成函数的形式，提高了复用率。接下来将函数应用在进行部分简单处理后的图像上，其结果如图 3-10（从原始二维码图片（左），去掉边缘以及 logo（中），最后将大小变成 25×25 的整个过程）所示：

```
    # 读图片，BGR 转换为 RGB
    img_jizhiQR = cv2.imread("./jizhi_qinding.png")
    img_jizhiQR = img_jizhiQR[:,:,::-1]

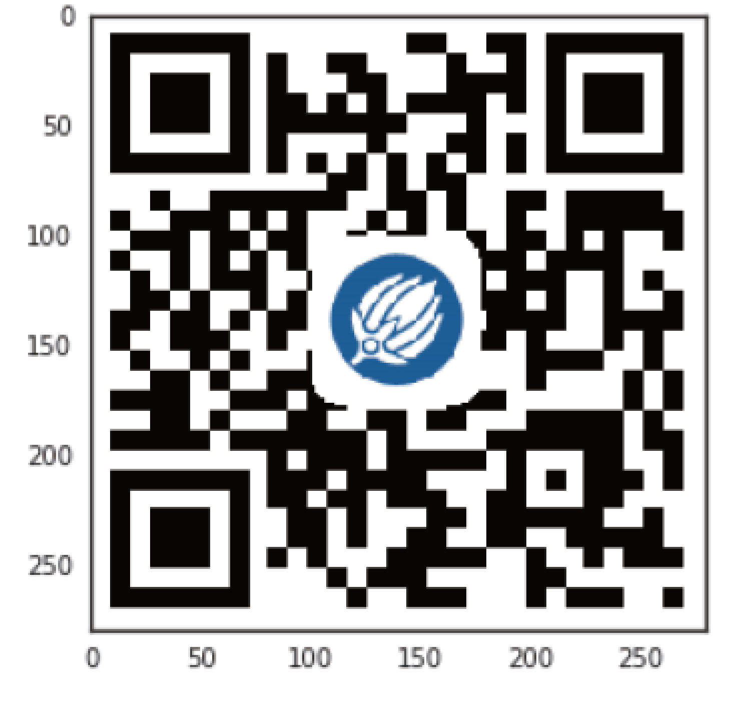
    # 去掉边缘的白色
    img_jizhiQR_gray = cv2.cvtColor(img_jizhiQR, cv2.COLOR_RGB2GRAY)
    np_totalRow = np.arange(img_jizhiQR.shape[0])
    idx_rowUsed = np_totalRow[img_jizhiQR_gray.mean(0) != 255]
    idx_colUsed = np_totalRow[img_jizhiQR_gray.mean(1) != 255]
    img_jizhiQR_rmBlank = img_jizhiQR[idx_rowUsed,:,:][:,idx_colUsed,:]

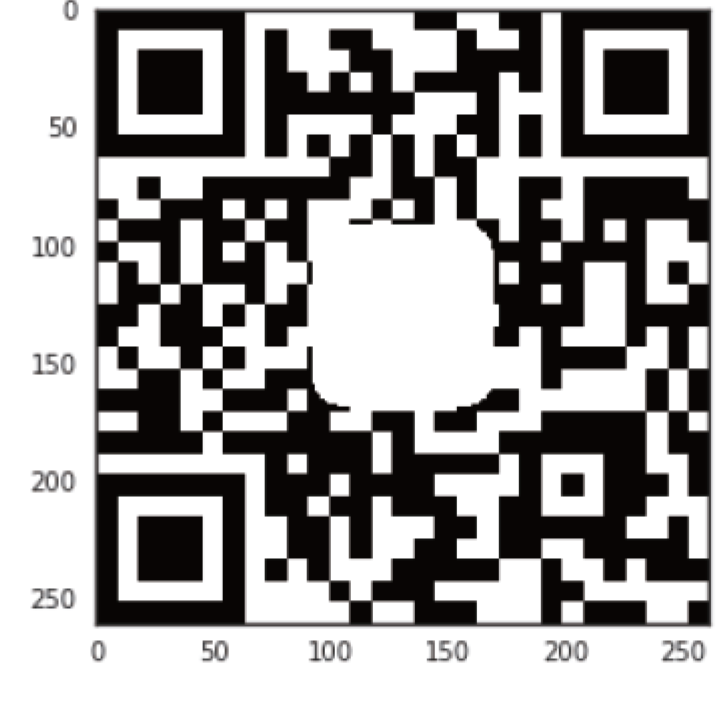
    # 去掉中间蓝色部分，即新建一个空白矩阵，然后将原先矩阵是黑色的部分在新矩阵中设为 255
    img_jizhiQR_new = np.zeros([img_jizhiQR_rmBlank.shape[0], img_jizhiQR_
rmBlank.shape[1]])
    idx_black = (img_jizhiQR_rmBlank[:,:,0]<10)*(img_jizhiQR_
rmBlank[:,:,1]<10)*(img_jizhiQR_rmBlank[:,:,2]<10)
    img_jizhiQR_new[idx_black] = 255

    # 用 tensorflow 池化函数，将二维码大小减少到 29×29
    np_25_counts_tf = AvgPool(img_jizhiQR_new, 29)

    fig = plt.figure(figsize=(15,5))
    ax1 = fig.add_subplot(131)
    ax2 = fig.add_subplot(132)
    ax3 = fig.add_subplot(133)

    ax1.imshow(img_jizhiQR)
    ax2.imshow(img_jizhiQR_new)
    ax3.imshow(np_25_counts_tf)
```





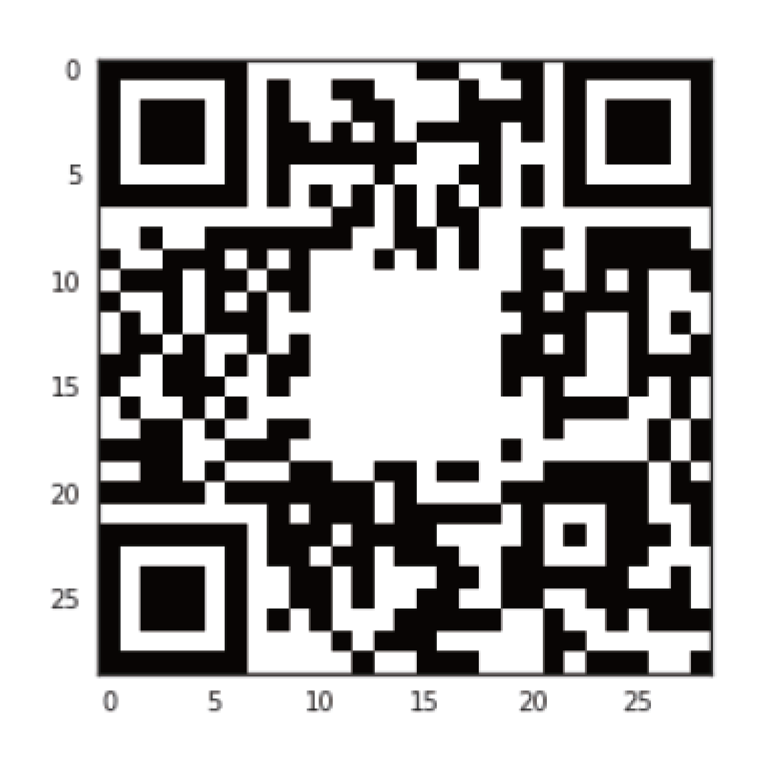


图 3-10 运行结果

3.3 使用 OpenCV“抠图”——基于颜色通道以及形态特征

3.2 节用简单的矩阵操作方法对二维码图片进行了一系列操作，包括去 Logo、降低像素等。本节谈一谈如何对图片进行一些比较复杂的操作，比如我们 3.1.1 节中用数组切片来抓取足球，使用的命令是：

```
ball = img[280:340, 330:390]
```

为什么是这个坐标？这个坐标是我们人眼看出来的，能否让计算机自动识别这个足球的位置？这里提供一种基于 OpenCV 的“套路”。

首先，还是需要认真观察数据，就是刚才抓下来的球（如图 3-11 所示）：

```
plt.imshow(ball)
```

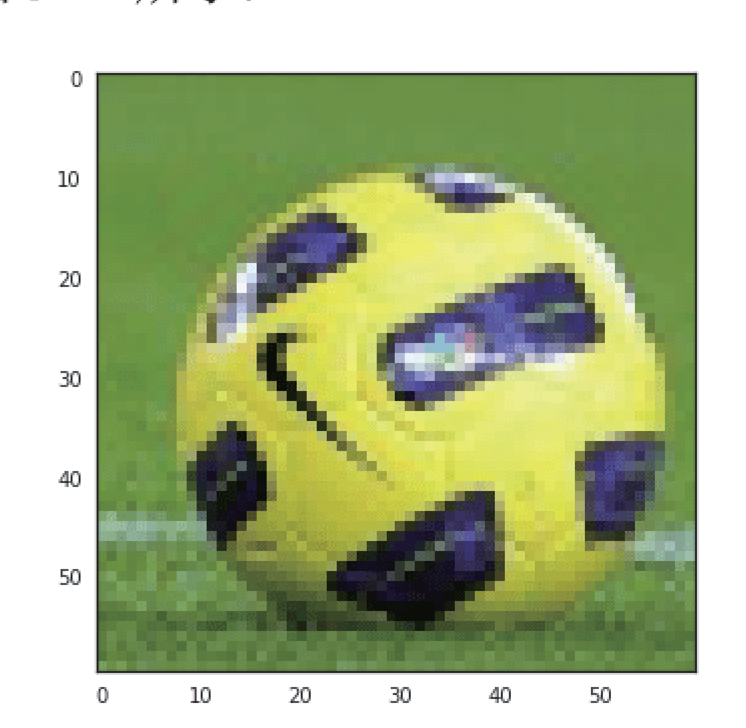


图 3-11　从原图中截取足球区域

这个球具有以下特点：

- 看起来是个圆形。
- 颜色是黄色＋藏蓝色。
- 由于在草地上，因此背景是绿色。

根据这三个特点，我们大致确定思路，几个步骤依次对应下面几个特点：

- 看看有没有圆形的识别算法——Hough 变换。
- 用黄色＋藏蓝色将球从背景提取出来。
- 用绿色过滤背景色。

我们观察这张球小型图片中的 60×60 =3600 个像素，其中 RGB 的分布如何。这里仿照第 2 章用过的 seaborn 的 pairplot 函数来表达 RGB 三种颜色的两两组合，其结果如图 3-12 所示：

```
fig = plt.figure(figsize=(10,10))
ax = fig.add_subplot(332)
for r in ball.reshape(-1, 3):
    ax.plot(r[1], r[0], '.', c=(r[0]/255., r[1]/255., r[2]/255.) )

ax = fig.add_subplot(333)
for r in ball.reshape(-1, 3):
    ax.plot(r[2], r[0], '.', c=(r[0]/255., r[1]/255., r[2]/255.) )

ax = fig.add_subplot(336)
for r in ball.reshape(-1, 3):
    ax.plot(r[2], r[1], '.', c=(r[0]/255., r[1]/255., r[2]/255.) )

for i,color in enumerate(['Red', "Green", "Blue"]):
    ax = fig.add_subplot(3,3,i*3+i+1)
    ax.text(5,5, color)
    ax.plot(0,0)
```

```
        ax.plot(10,10)
        ax.set_xlim(0, 10)
        ax.set_ylim(0, 10)
        ax.axis('off')
```

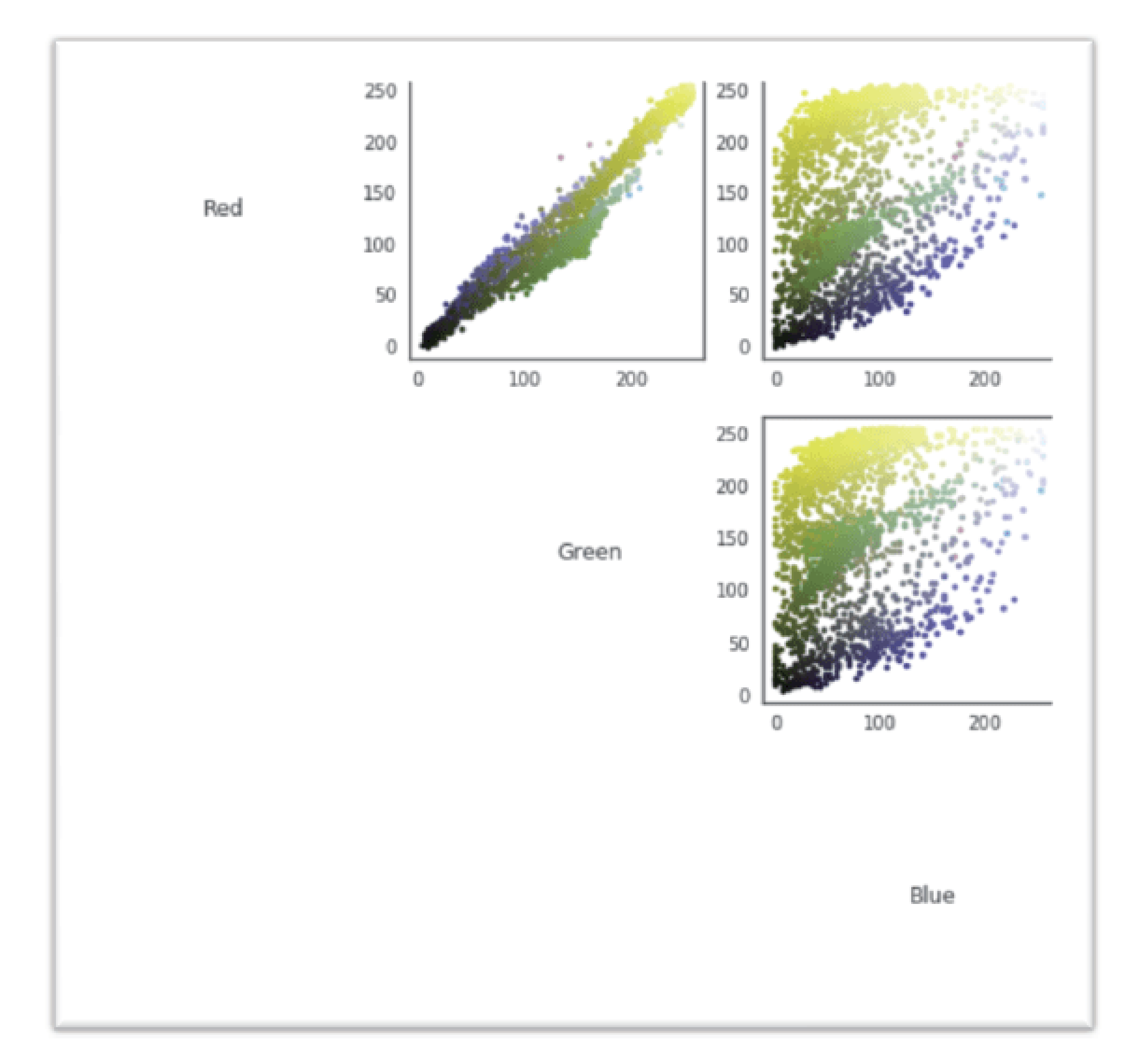


图 3-12 图 3-11 中各像素点 RGB 的分布情况

图中的黄色与蓝黑色均为足球的颜色，而绿色则是足球场的颜色。我们接下来要做的就是找一个规则来区分足球与足球场。这里简单地写一个规则组合，其结果如图 3-13 所示：

```
    fig = plt.figure(figsize=(15, 5))
    ax1 = fig.add_subplot(131)
    ax2 = fig.add_subplot(132)
    ax3 = fig.add_subplot(133)
    ax1.imshow( (ball[:,:,0]>200) )
    ax2.imshow( (ball[:,:,0]>130)+(ball[:,:,0]<50) )
    ax3.imshow( (ball[:,:,0]>130)+(ball[:,:,0]<50)+(ball[:,:,1]<120) )
```

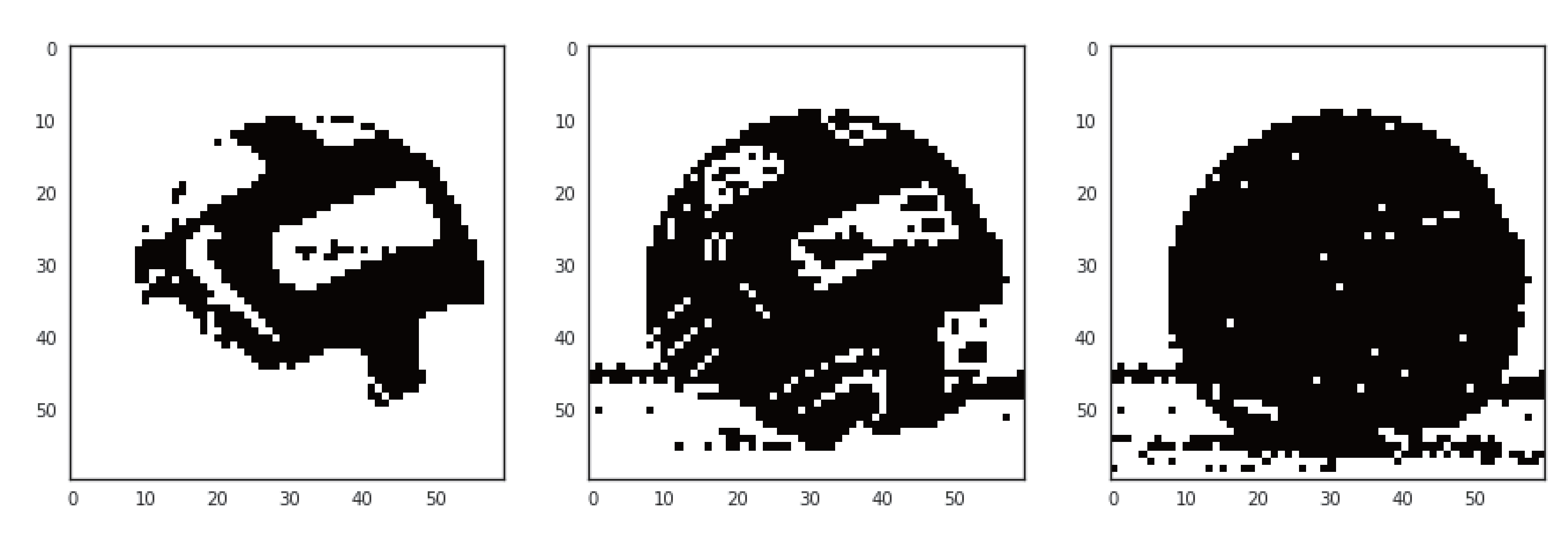


图 3-13 使用颜色规则对球的区域进行二值化（黑色代表球的区域）

好了，根据颜色规则，我们已经依稀得到了一个圆形物体。下一步，将这一规则推广到整张图片中，其结果如图 3-14 所示。

```
fig = plt.figure(figsize=(12, 5))
ax1 = fig.add_subplot(121)
ax2 = fig.add_subplot(122)

img1 = ((img[:,:,0]>130)+(img[:,:,0]<50)+(img[:,:,1]<120)).astype(np.uint8)
ax1.imshow(img)
ax2.imshow(img1)
```

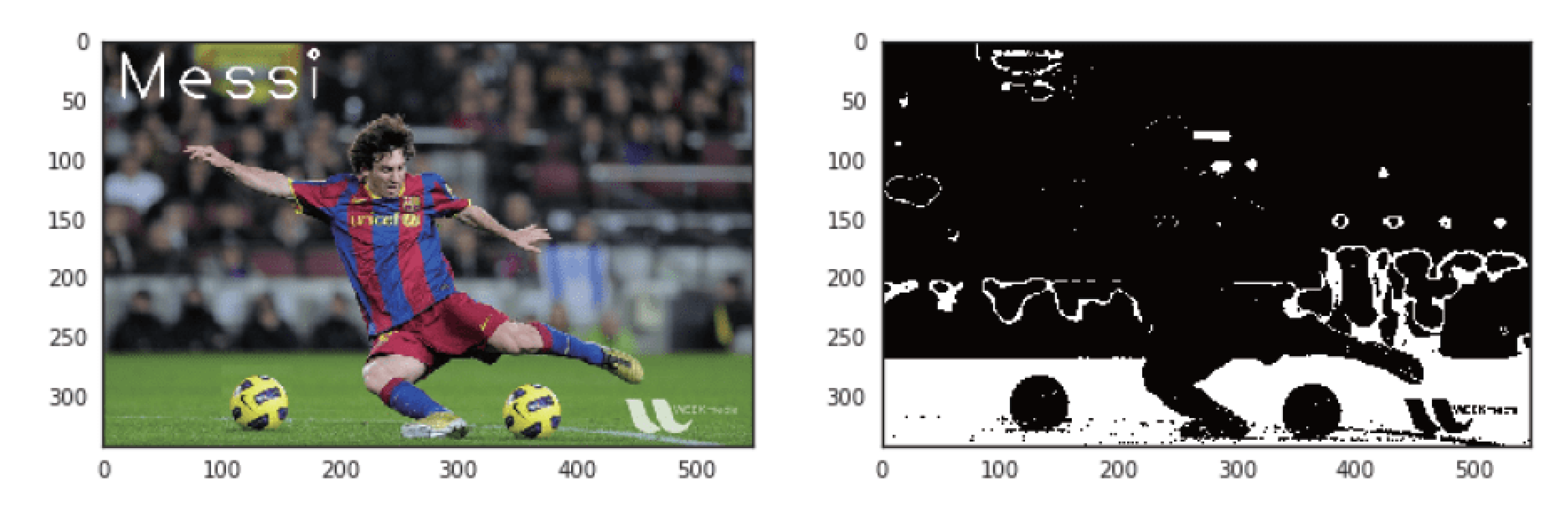


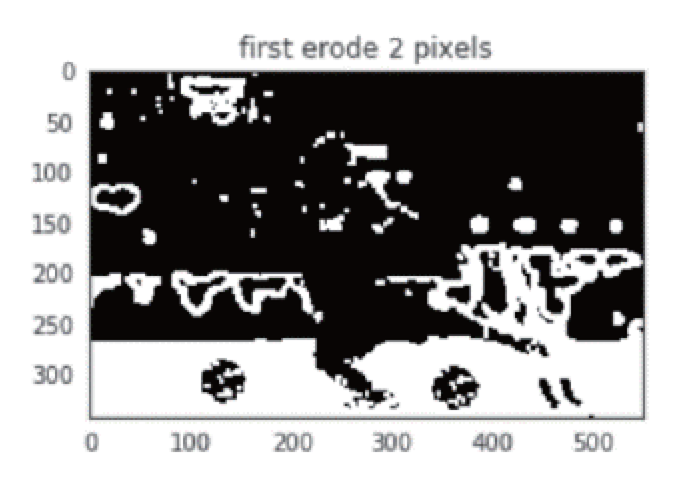
图 3-14 使用颜色规则对整张图片球的区域进行二值化（黑色代表球的区域）

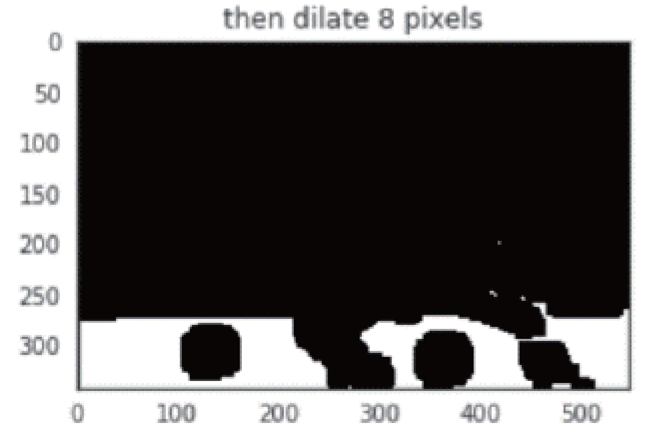
由此可见，对整张图片进行的颜色二值化处理还是比较有效的，至少球这里出现了圆形的形状，我们经过简单的处理后，将这里的圆形提取出来即可。但是这里任务并不轻松，因为图中还是比较杂乱：首先，球中间的黑色部分含有白色，需要填充掉；其次，背景部分有很多观众、标语造成的白色环状噪点，这些噪点不大，但会在圆形检测环节干扰结果。

为了让图片结果减少噪点，我们这里利用 OpenCV 进行一组侵蚀（erode）稀释（dilute）操作，如图 3-15（对图 3-14 的结果先进行 2 像素侵蚀，再进行 8 像素的稀释，最后进行 3 像素的侵蚀的整个过程）所示。形象地说，可以将图中的黑色区域看成是海岛，白色看成是海洋。进行侵蚀操作后，海水上涨，白色区域变多，将孤立的黑色区域“淹没”；然后进行稀释操作。此时海水退去，没有被完全淹没的、面积较大的海岛基本恢复以前的样子，而被完全淹没的、面积小的海岛则从此消失。

```
kernel = cv2.getStructuringElement(cv2.MORPH_RECT,(3, 3))

img_e   = cv2.erode(img1, kernel, iterations=2)
img_de  = cv2.dilate(img_e, kernel, iterations=8)
img_ede = cv2.erode(img_de, kernel, iterations=3)
fig = plt.figure(figsize=(15, 5))
ax1 = fig.add_subplot(131)
ax2 = fig.add_subplot(132)
ax3 = fig.add_subplot(133)
ax1.imshow(img_e)
ax2.imshow(img_de)
ax3.imshow(img_ede)
ax1.set_title(u"first erode 2 pixels")
ax2.set_title(u"then dilate 8 pixels")
ax3.set_title(u"finally erode 3 pixels")
```





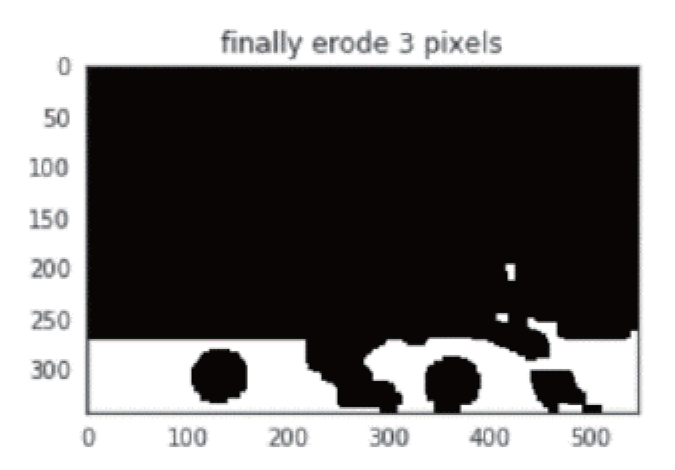


图 3-15 侵蚀、稀释操作

经过侵蚀、稀释操作后，图中的黑色部分将会完全连在一起，面积较小的噪点则会被完全淹没在背景中。此时，下面的两个黑色圆形物体正是我们需要识别的球。现在先不着急，因为 OpenCV 的圆形识别算法识别的并非圆形物体，而是圆形的线条。因此还需要将这张图片的边缘提取出来，提取边缘的方法就是对整张图片计算梯度，此时如果一张像素周围全是黑色或者全是白色，则梯度为 0；而旁边既有黑色又有白色，则会产生一个颜色梯度，即图片边缘。我们可以用 Sobel 算子分别对图像的横向、纵向计算颜色梯度，然后求平方根，得到总梯度。Sobel 算子是一个 3×3 的卷积核，定义如下：

$$Sobel_kernel_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}$$

$$Sobel_kernel_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix}$$

计算过程如下：

$$Sobel_x = Image * Sobel_kernel_x$$

$$Sobel_y = Image * Sobel_kernel_y$$

Opencv-python 的对应代码如下：

```
sobelx = cv2.Sobel(img_ede*255, cv2.CV_64F, 1, 0)
sobely = cv2.Sobel(img_ede*255, cv2.CV_64F, 0, 1)
img_sob = np.sqrt(sobelx**2+sobely**2).astype(np.uint8)
plt.imshow(img_sob)
```

其结果如图 3-16 所示。

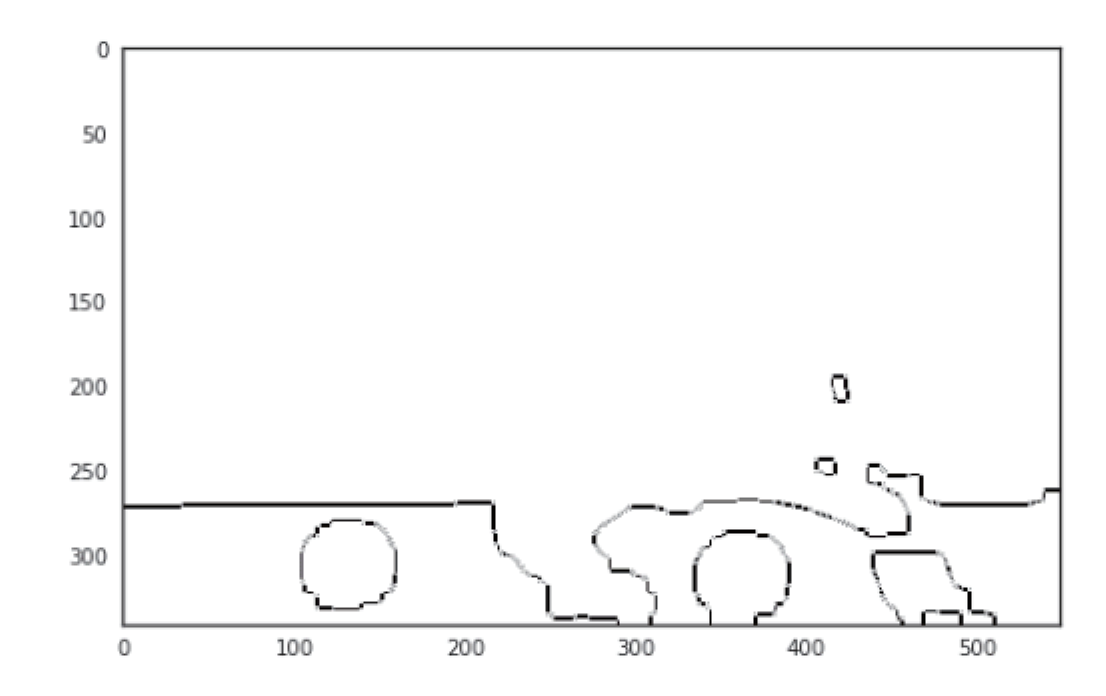


图 3-16 对图 3-15 的结果使用 Sobel 算子找出其边缘位置所在

此时已经得到了圆形的边缘，接下来调用 OpenCV 的圆形检测函数——cv2.HoughCircle，用 Hough 变换提取图中的圆形物体即可，如图 3-17 所示。

```
gray = img_sob
# 首先用 Canny 算子过滤边缘
canny = cv2.Canny(gray, 200, 300)

# 其次用中位数进行卷积操作，平滑颜色梯度
gray = cv2.medianBlur(gray, 5)

# 最后用 HoughCircle 函数检测圆形物体。这里
np_hc = cv2.HoughCircles(gray, cv2.HOUGH_GRADIENT, dp=1, minDist=60,
              param1=200,
              param2=10,
              minRadius=20,
              maxRadius=30)

# 展示结果
fig = plt.figure()
ax = fig.add_subplot(111)
ax.imshow(img_sob, "gray")
img_tmp = np_hc
for i in range(np_hc.shape[1]):
    img_tmp = cv2.circle(img, (np_hc[0,i,0],np_hc[0,i,1]), np_hc[0,i,2],
(255, 0, 0), 8)

plt.imshow(img_tmp)
```



图 3-17　成功识别图 3-14 中球所在的位置

最后多说几句：

（1）Hough 变换可以在图中检索线段以及圆形，具体原理与利用极坐标系来表示线段、圆上的点有关，大家可以另行学习。常见的应用场景包括自动驾驶系统检测道路线（直线检测），以及显微镜下观察细胞（圆形检测）。

（2）通常情况下，Hough 变换可以直接用在灰度图上，即上一个代码框里的 gray = img_sob 可以是 gray = img_gray。这里之所以没有这么做，是因为背景部分同样有很多类似圆形的物体，造成了很多干扰，可能需要调试很多参数才能得到想要的结果，读者不妨试一试。如果通过颜色选择将特征二值化，进而通过侵蚀、稀释操作进一步去除背景噪声之后，可以让输入的图形更加简单，更加方便参数的调试。

3.4 基于传统特征的传统图像分类方法

3.3 节介绍了如何使用图片的颜色以及形状特征，在图像中标注足球的位置。这个标注过程实际上还是基于人工规则进行的，存在很多问题：

- 只能识别蓝黄相间的球。
- 只能识别绿色草地上的球。
- 摄像机距离拉远、拉近后球的像素大小改变，仍然无法识别。

由此可见，人工规则在实际应用的场景中会遇到各种预想不到的结果，在这些情况下，人工规则的表现会大打折扣。为了解决这些问题，图像处理技术引入了机器学习方法，希望通过机器规则来代替人工规则。这一思（tao）路（lu），总体如下：

（1）针对某一图片，将几百、上千像素图片简化为少数几个区域，计算每个区域中轮廓特征的走向。

（2）将正负样本所有图片执行第三步，将轮廓走向图放入机器学习分类器进行训练。

（3）将训练好的分类器应用在新的图片中。

我们审视这一思路，发现这个过程的核心思想其实是在**简化图片**——第一步中，原本图片里各种不同的物体被减少到只剩下轮廓了，但这还不够，最后计算的是一个区域内的轮廓走向，此时一张 1200×800×3 大小的图片，可能只剩下 1000 个点了，信息减少了上千倍。

这么做的原因首先是传统的机器学习分类器，对输入数据的大小有一个大致的上限，几千张图像数据如果不经过简化，直接送入模型，模型是无法训练输入数据的；其次，图像数据最有趣的一点是图中一个像素点和周围像素点周围联系非常紧密，数据的冗余度很大，这种冗余体现在很多时候，即使给图片压缩、涂黑、打马赛克，我们也大致知道这张图片的内容；同理，对图片进行简化，也是基于这一思想。

本节基于这个思路来介绍一下如何用 python-opencv 处理图像，对图片进行简化处理，然后用上一章的机器学习方法进行图像分类。我们使用优达学城（Udacity）自动驾驶纳米学位的一个开源项目一部分内容作为讲解材料，希望了解完整部分的读者可以去优达学城官网 udacity.cn 以及开源地址 https://github.com/udacity/CarND-Vehicle-Detection 查看完整内容。

我们使用了优达学城标注的一个行车记录仪数据集，这个数据集来自 GTI 以及 KITTI 数据集，标注了车辆以及非车辆的情况，具体如下：

```
    # 下载数据集地址
    # 65.4MB    https://s3.amazonaws.com/udacity-sdc/Vehicle_Tracking/
vehicles.zip
    # 54.9MB    https://s3.amazonaws.com/udacity-sdc/Vehicle_Tracking/non-
vehicles.zip

    # 观察数据集
    import pandas as pd
    import seaborn as sns
    import numpy as np
    import matplotlib.pyplot as plt
    import sys
    import cv2
    import scipy.stats as stats
    from tqdm import tqdm
    import os
    from sklearn.metrics import confusion_matrix, auc, roc_auc_score
    import matplotlib.pyplot as plt
    import sklearn
    from skimage.feature import hog
    import pandas as pd
    import os
    from tqdm import tqdm
    from sklearn.model_selection import train_test_split
    from sklearn.svm import LinearSVC,SVC
    from sklearn.preprocessing import StandardScaler
    import time
    from scipy.ndimage.measurements import label
    import numpy as np
    import functools
    import pickle

    l_samp = !ls ./dataset/*vehicles/*/*

    M_ClassDict = {"non-vehicles" : 0, "vehicles" : 1}
    pd_SampClass = pd.DataFrame({
        "Sample" : l_samp,
        "Class"   : list(map(lambda x: M_ClassDict[x], list(map(lambda x:
x.split("/")[2], l_samp))))
    })[['Sample', 'Class']]
```

```
    pd_SampClass_train,pd_SampClass_cv = train_test_split(pd_SampClass,
test_size=0.33, random_state=42)
    pd_SampClass_train.head()
```

运算结果：

	Sample	Class
6490	./dataset/non-vehicles/GTI/image2279.png	0
2736	./dataset/non-vehicles/Extras/extra3857.png	0
15541	./dataset/vehicles/KITTI_extracted/4374.png	1
1068	./dataset/non-vehicles/Extras/extra223.png	0
3660	./dataset/non-vehicles/Extras/extra4821.png	0

```
    fig = plt.figure(figsize=(12, 6))
    for i in range(5):
        image = cv2.imread(pd_SampClass_train['Sample'].iloc[i])
        image = image[:,:,::-1]
        ax = fig.add_subplot(1,5,i+1)
        ax.imshow(image)
        ax.set_title(pd_SampClass_train['Class'].iloc[i])
```

其结果如图 3-18 所示。

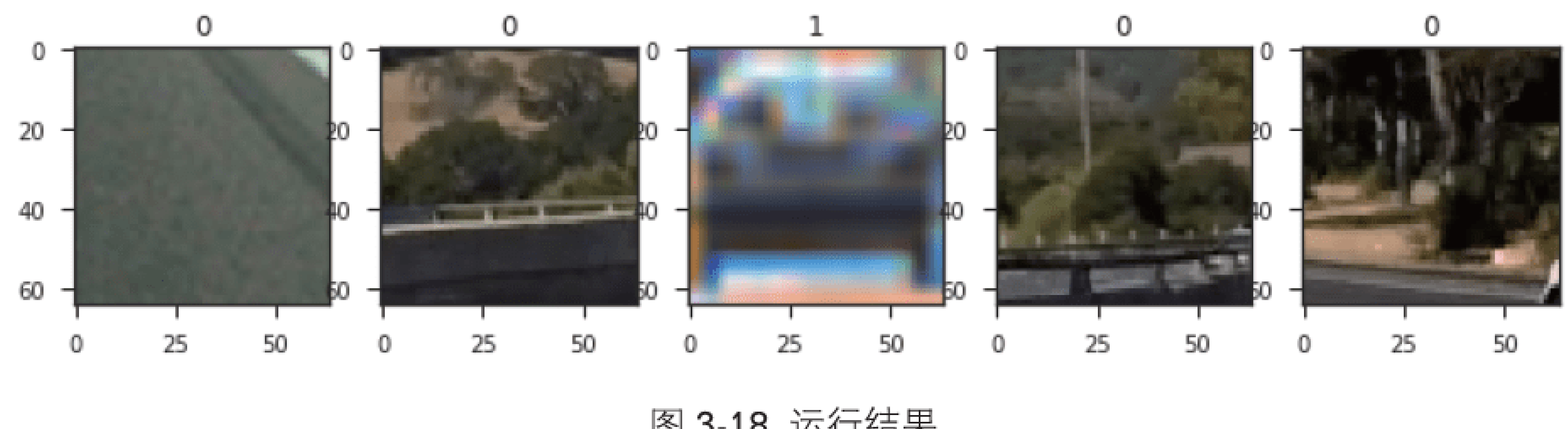


图 3-18 运行结果

我们的任务是利用几千张这样的标注图片组成的数据集来训练一个分类器，以区别输入图片是否是车辆。

3.4.1 将图片简化为少数区域并计算每个区域轮廓特征的方向

这里其实是运用方向梯度直方图算法（Histogram of Oriented Gradients，HOG）来实现的。这一算法可以分为以下步骤（http://www.learnopencv.com/histogram-of-oriented-gradients/）：

- 图像归一化（可选）。
- 计算图像中 x 以及 y 方向的梯度。
- 根据梯度，计算梯度柱状图。
- 对块状区域进行归一化处理。

- 展开结果，将一张图片转换成一个一维的特征向量。
- 提取的特征，交给支持向量机或神经网络分类器进行训练分类。

例如，可以将如图 3-19 所示的左图作为 HOG 算法的输入文件，得到类似右图的结果。首先，HOG 算法的结果只表示轮廓线的变换，无关颜色（如白色的背景以及右下角黑色部分）都没有轮廓线；其次，整张图 x、y 方向有成百上千的像素点，这里被缩小到了 32×32 的网格区域，这一点类似前面提到的二维码的读取。当然，这里与二维码不同的是，除了分组求平均值以外，还考虑了轮廓线的方向性，即线条颜色深浅表示了轮廓线数目的多少，同时线条的方向也可以表示这块区域内轮廓线的总体走势。



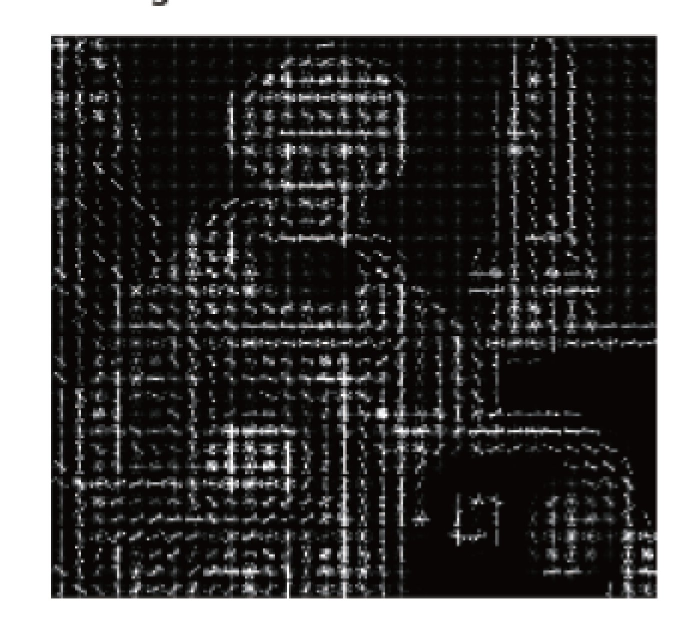


图 3-19 HOG 算法实例（图片来自 skimage 的官方文档）

进一步阅读官方文档，发现用法如下：

```
    fd, hog_image = hog(image, orientations=8, pixels_per_cell=(16, 16),
cells_per_block=(1, 1), visualize=True)
```

这里有三个可以调的参数，即 orientations、pixels_per_cell 以及 cells_per_block。这三个参数的含义简单说明如下：

- pixels_per_cell 是指多少个像素作为一个网格来计算，这个值越高，切出来的网格就越少，整个 HOG 的结果就越粗略。
- orientation 是指切出来每个网格中有几种方向的走势，如果是四种，就是上、下、左、右；如果是八种，就再增加上左、上右、下左、下右四种方向。
- cells_per_block 是指一个网格中使用几个方向指针，如果是十字交叉的情况，则至少需要两个方向指针进行交叉，才可以表示出十字。

然后就是借助 matplotlib 可视化包查看一下不同参数的结果，如图 3-20 所示。

```
fig = plt.figure(figsize=(20, 10))

img = cv2.imread("./dataset/vehicles/GTI_Far/image0000.png")
img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

for i1,pix_per_cell in enumerate([6,8,10]):
    for i2,cell_per_block in enumerate([2,3]):
```

```
            for i3,orient in enumerate([6,8,9]):
                features, hog_image = hog(img_gray, pixels_per_cell=(pix_
per_cell,pix_per_cell),
                cells_per_block=(cell_per_block,cell_per_block),
orientations=orient, visualise=True, feature_vector=False
                )
                ax = fig.add_subplot(3,6,i1*6+i2*3+i3+1)
                ax.imshow(hog_image, 'gray')
                ax.set_title("Pix%d_C%d_Ori%d" % (pix_per_cell, cell_per_
block, orient))
```

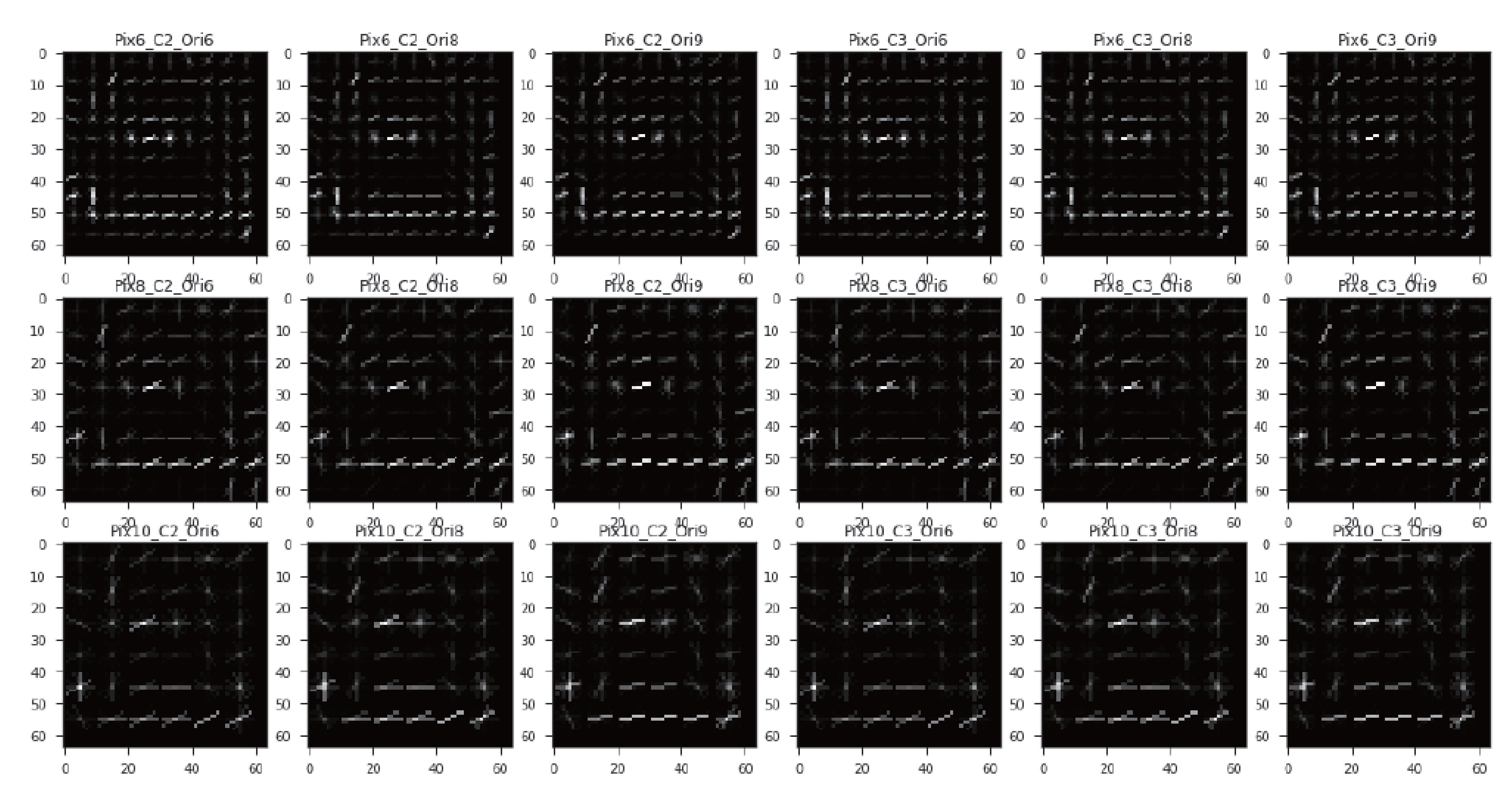


图 3-20 使用不同的 HOG 参数分析输入图片得出不同的结果

这里决定用 cell_per_block = 2，因为感觉结果中最多只出现了两个方向的交叉。然后 orientations = 9 对拐角处的特征保留得更好，看起来像车的形状。最后，pix_per_cell = 8 看起来正好可以保持车的形状，取 6 得到的网格过多，取 10 则得到的网格过少。

3.4.2 将 HOG 变换运用在所有正负样本中

这里将正负样本抽样相关的函数写成一个 class，然后在这里引用 class 进行相应的操作，得到正负样本在各个图片中的区域。然后从这些区域中提取图像文件，resize 到 64×64，计算 HOG 值，进而保存在矩阵中。

注意，下一步需要利用机器学习进行相应的判别，为了评价分类的准确性，这里需要将正负样本进一步切割为训练集和测试集。

```
    # 这里只看灰度图的轮廓，不考虑颜色。如果需要考虑，这里可以继续添加
    l_colorSpace = [cv2.COLOR_BGR2GRAY]
    l_names = ["GRAY"]
```

```
    l_len    = [1]

    def get_hog_features(img, orient, pix_per_cell=8, cell_per_block=2,
                                vis=False, feature_vec=True):

        if vis == True:
            features, hog_image = hog(img, orientations=orient, pixels_per_
cell=(pix_per_cell, pix_per_cell),
                cells_per_block=(cell_per_block, cell_per_block), transform_
sqrt=True,
                visualise=vis, feature_vector=feature_vec
            )
            return features, hog_image

        else:
            features = hog(img, orientations=orient,
                    pixels_per_cell=(pix_per_cell, pix_per_cell),
                  cells_per_block=(cell_per_block, cell_per_block),
                    transform_sqrt=True,
                  visualise=vis, feature_vector=feature_vec
    )
            return features

    def get_features(img, pix_per_cell=8,cell_per_block=2,orient=9,
getImage=False, inputFile=True, feature_vec=True):
        l_imgLayers = []
        for cs in l_colorSpace:
            if inputFile:
                l_imgLayers.append(cv2.cvtColor(cv2.imread(img), cs))
            else:
                l_imgLayers.append(cv2.cvtColor(img, cs))

        l_hog_features = []
        l_images = []
        for feature_image in l_imgLayers:
            hog_features = []
            n_channel = 1
            if len(feature_image.shape) > 2:
                n_channel = feature_image.shape[2]
            for channel in range(n_channel):
                featureImg = feature_image
                if n_channel > 2:
                    featureImg = feature_image[:,:,channel]

                vout,img = get_hog_features(featureImg,
                                    orient, pix_per_cell, cell_per_block,
```

```
                                     vis=True, feature_vec=feature_vec)
                if getImage:
                    l_images.append(img)
                #print(featureImg.shape, vout.shape)
                hog_features.append(vout)

            l_hog_features.append(list(hog_features) )

    if getImage:
        return l_images
    else:
        return functools.reduce(lambda x,y: x+y, l_hog_features)

# 对划分好的训练集、测试集提取图像信息，计算 HOG 值，存储中间结果
if os.path.isfile("./X_train.npy") == 0:
    l_X_train = []
    l_X_test  = []
    for r in tqdm(pd_SampClass_train.iterrows()):
        l_X_train.append(
             np.array(get_features(r[1]['Sample'])).ravel()
        )

    for r in tqdm(pd_SampClass_cv.iterrows()):
        l_X_test.append(
             np.array(get_features(r[1]['Sample'])).ravel()
        )

    X_train = np.array(l_X_train)
    X_test  = np.array(l_X_test)
    np.save("./X_train.npy", X_train)
    np.save("./X_test.npy",  X_test)
else:
    X_train = np.load("./X_train.npy")
    X_test  = np.load("./X_test.npy")

y_train = pd_SampClass_train['Class'].values
y_test  = pd_SampClass_cv['Class'].values
```

3.4.3 训练模型

完成前面的步骤后将开始模型的训练。训练的第一步是需要对数据进行标准化处理：

```
X_scalerM = StandardScaler()
X_trainT = X_scalerM.fit_transform(X_train)
X_testT  = X_scalerM.transform(X_test)

X_trainTs,y_trainTs = sklearn.utils.shuffle(X_trainT, y_train)
```

使用 sklearn 的支持向量机模块进行训练。训练后，看看验证集的表现：

```
svc = SVC(random_state=0, C=1)
t=time.time()
svc.fit(X_trainTs, y_trainTs)
t2 = time.time()
print(round(t2-t, 2), 'Seconds to train SVC...')
#51.54 Seconds to train SVC...
```

训练完成。接下来，用划分出来的验证集查看模型的表现。

3.4.4 将训练好的分类器运用在新的图片中

开始划分数据时就使用 train_test_split(pd_SampClass, test_size=0.33, random_state=42)，分别划分了训练集以及验证集。其中，66% 的数据被划分为训练集，用于上一步的模型训练。

这里，将用剩下的 33% 作为验证集来检验模型的表现。首先看准确率：

```
print('Test Accuracy of SVC = ', round(svc.score(X_testT, y_test), 4))
```

运行结果：

```
# out:
Test Accuracy of SVC =  0.9869
```

98% 的分类准确率还是不错的。接下来选十个样本看结果：

```
n_predict = 10
print('My SVC predicts: ', svc.predict(X_testT[0:n_predict]))
```

运行结果：

```
# out:
My SVC predicts:  [1 1 1 0 0 0 0 1 1 1]
```

```
print('For these',n_predict, 'labels: ', y_test[0:n_predict])
```

运行结果：

```
# out:
For these 10 labels:  [1 1 1 0 0 0 0 1 1 1]
```

选的前十个测试样本，结果都是预测正确。最后看 AUC 值：

```
pred = svc.predict(X_testT)
print("AUC for Merge dataset = %1.2f,\n" % (roc_auc_score(pred, y_test)))
```

运行结果：

```
# out:
AUC for Merge dataset = 0.99,
```

```
print(confusion_matrix(pred, y_test))
```

运行结果：

```
# out:
[[2929   55]
 [  22 2855]]
```

发现大多数预测准确，假阳性的数量是真阴性的 2 倍多，是一个基本可用的模型。至于这个模型的适用性如何，读者可以在网上找图片，resize 到 64×64，看看是否可以成功预测。

最后，总结本章所学的内容。本章开始部分提到，要用机器学习方法分析传统图像数据，可以有三种思路：

- 手动提取重要的特征，用数字表示。如鸢尾花数据集，当年就是用尺子量出来的长度、宽度，交给机器学习分类器。
- 用简单的图像处理操作，将图片转换为少数几个简单的轮廓特征，交给机器学习分类器。
- 用深度神经网络，让深度学习模型自动提取图片的各种特征，再用模型自动提取的特征训练分类器。

本章介绍的是第二种方法，即提取简单特征，交给机器学习分类器，如支持向量机。这种方法的问题想必读者也有切身体会：

（1）“笨”——连足球都不认识，需要手动提取颜色、轮廓，然后过滤噪声，最后交给霍夫变换检测器，计算机才认识这是个圆形。

（2）“眼花”——我们拿到的是一个高像素的图片，需要将图片的信息不断模糊、减少信息量，机器学习模型才能“认识”这张图片，才能用这种简化后的图片进行模型训练。

我们再思考一下，为什么传统计算机模型会显得很“笨”，而且眼神不好？一个很重要的原因是，传统的机器学习模型缺乏特征组合能力，尤其是对图像输入，计算机可以理解单独的一个像素，但是把单一像素与周围三五个点一起考虑，计算机模型在组合的时候，似乎不太能把握这一组点的关系，所以我们会用 opencv skimage 把人类在识别诸如球体、车辆这样的物体的关键因素提取出来，然后将提取的信息交给机器学习模型。

接下来介绍的基于深度神经网络的方法，特征提取部分就并不完全由人手动完成，计算机模型可以帮助数据分析者提取诸如球形、车体框架这样的特征。于是我们发现，相比传统的图像处理技术，计算机不“笨”了、不需要人提取特征了，眼神也变好了，可以直接识别原始图片了。这也是深度学习技术总是和人工智能相提并论的一个很重要的原因。

3.5 参考文献及网页链接

［1］ Getting Started with Images¶. Getting Started with Images — OpenCV-Python Tutorials 1 documentation. Available at: http://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_gui/py_image_display/py_image_display.html.

［2］ Histogram of Oriented Gradients. Histogram of Oriented Gradients — skimage v0.14dev docs. Available at: http://scikit-image.org/docs/dev/auto_examples/features_detection/plot_hog.html.

［3］ Hough Circle Transform. Hough Circle Transform — OpenCV 2.4.13.3 documentation. Available at: http://docs.opencv.org/2.4/doc/tutorials/imgproc/imgtrans/hough_circle/hough_circle.html.

［4］ Mallick, S. Home. Learn OpenCV (2016). Available at: http://www.learnopencv.com/histogram-of-oriented-gradients/.

［5］ Sobel Derivatives. Sobel Derivatives OpenCV 2.4.13.3 documentation. Available at: http://docs.opencv.org/2.4/doc/tutorials/imgproc/imgtrans/sobel_derivatives/sobel_derivatives.html.

［6］ Udacity. udacity/CarND-Vehicle-Detection. GitHub (2017). Available at: https://github.com/ udacity/CarND-Vehicle-Detection.

［7］ tf.estimator Quickstart ｜ TensorFlow. TensorFlow. Available at: https://www.tensorflow.org/ get_started/estimator.

第 4 章

继往开来——使用深度神经网络框架

通过第 2、3 章内容的学习，读者对于机器学习以及图像处理的基本概念应该有了大致的了解。从本章内容开始，我们将逐步开始介绍深度学习相关的内容，本章内容将简要介绍深度学习“可微分编程”的基本框架，然后介绍如何用简单的代码实现这一框架。

4.1 从逻辑回归说起

第 2 章提到传统机器学习算法时，就提到了逻辑回归算法。

（1）随机初始化一组 ω，比如可以全设为 0，当然实际上不推荐这样。

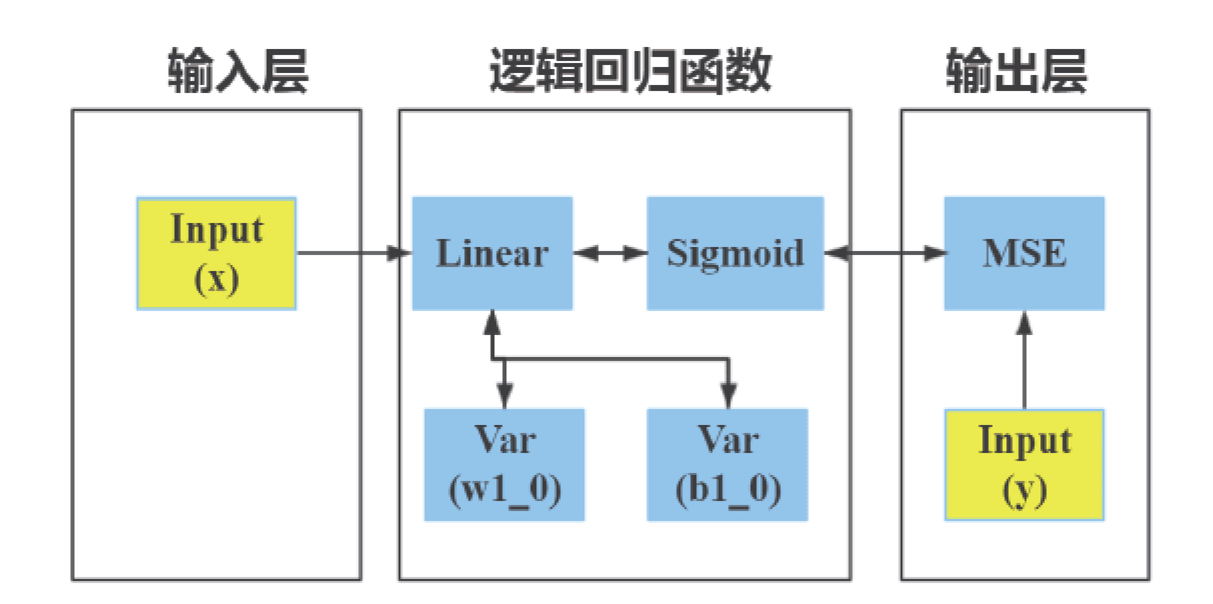
（2）训练集中，逻辑回归函数里，输入特征 x，计算 ωTx，得到预测结果 $\hat{y}$。

（3）计算全部训练集中逻辑回归的结果 $\hat{y}$ 和实际 y 的差别。

（4）根据上一步的差别更新 ω。

（5）重复（2）~（4）若干次（iterations）。

这个算法，可以用如下简单的框架表示：

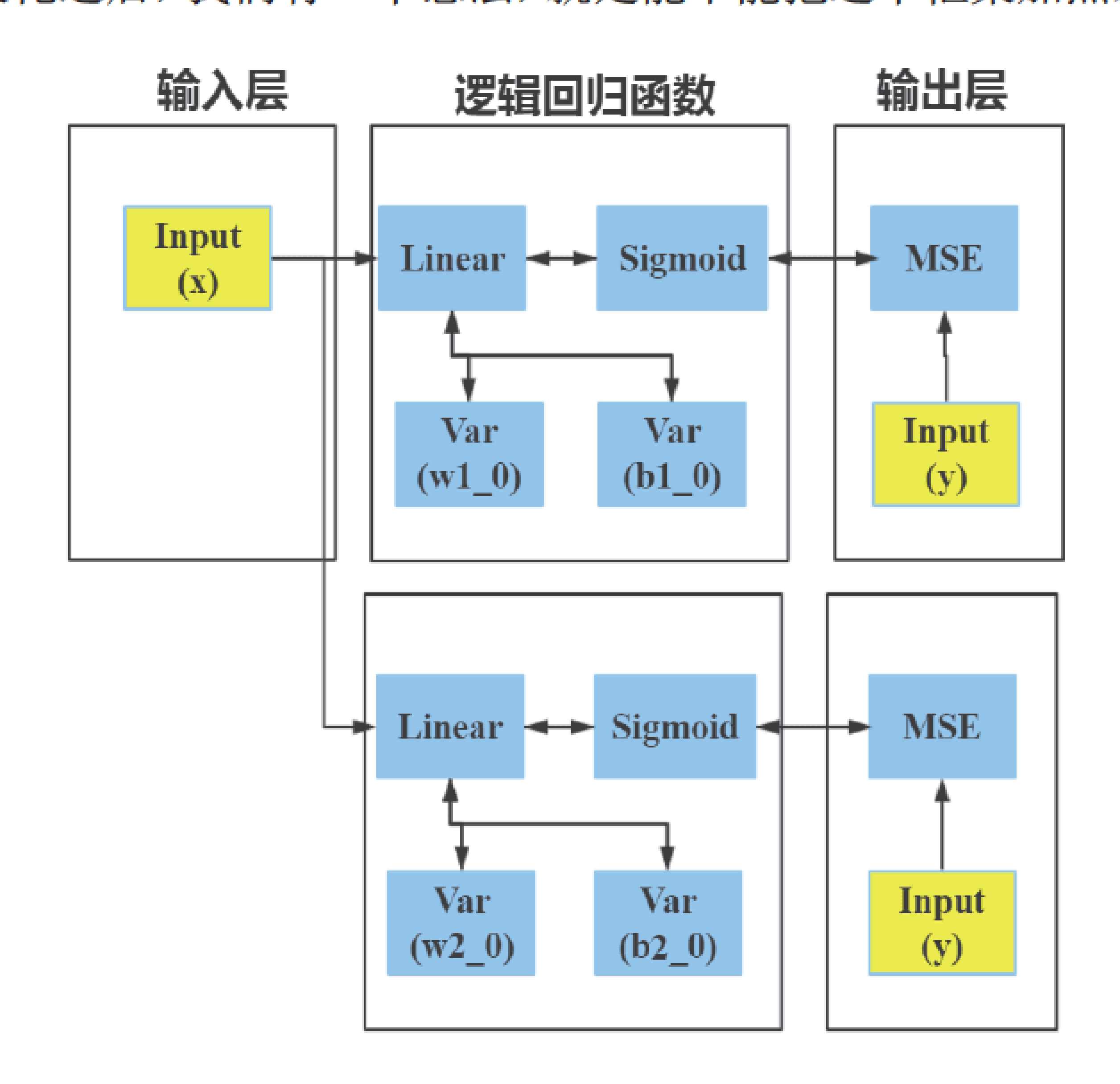


注意，在第 2 章的代码中，为了省事，将这里的 x 输入换成了 [x, 1]，加了一个维度，此时两个输入 ω、b 就合并成了一个新的 ω。

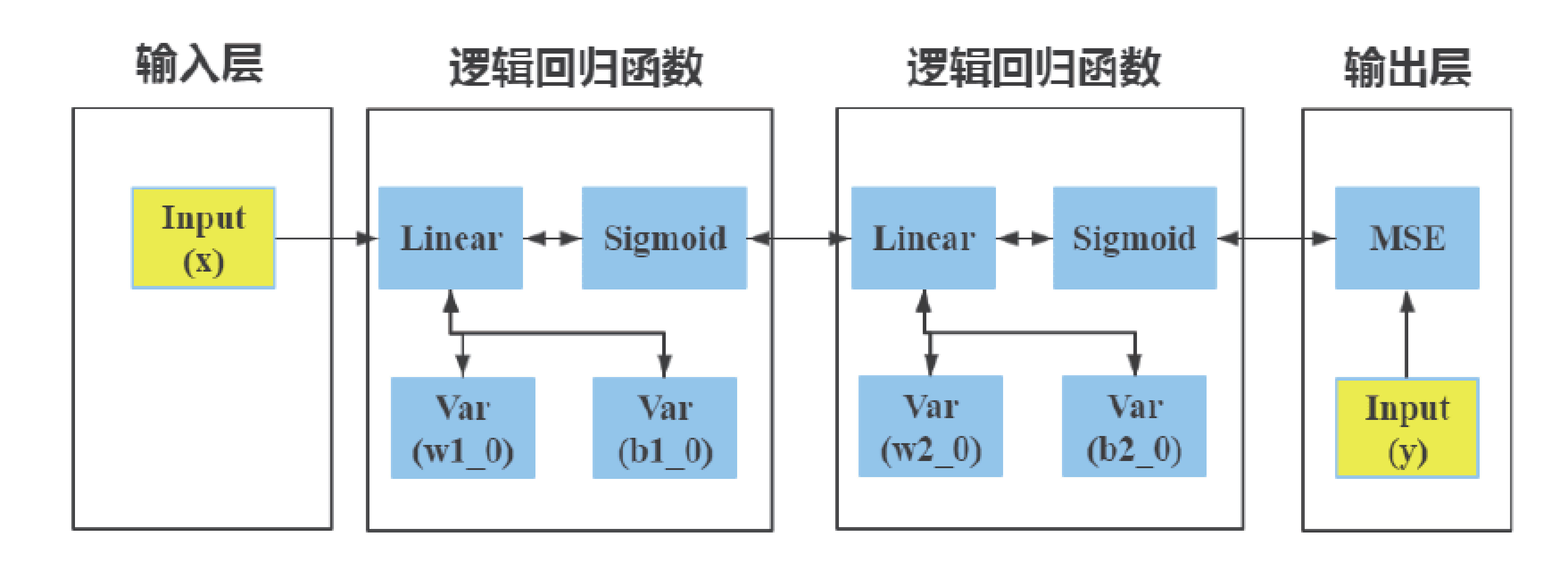
其次图中的双箭头代表了两个过程：

- 从左到右、从下到上的箭头代表了算法第二步得到预测结果的过程（MSE 处与 y 比较除外，是第三步）。
- 从右往左、从上往下的箭头代表了算法第四步中更新算法 ω、b 的过程。

这里将算法框架化之后，我们有一个想法，就是能不能把这个框架加点东西，比如变成这样：



或者这样：



当然，读者还可以继续思考，设计新的框架。这里，以上两种模型成了逻辑回归的“并联”“串联”形式。其中，“并联”形式类似民主投票，即可以训练多个逻辑回归模型，每个模型给一个测试样本预测一个结果，然后多个模型汇总结果，比如 70% 的模型都通过这个样本属于某个分类，则这个结果就被预测成这种分类。这种思路逐渐发展成了模型的聚合（Ensemble）方法，即通过多个弱分类器进行组合，形成一个强分类器。我们在第 9、10 章时使用的模型融合策略就是基于这种思想。这部分更多的内容，有兴趣的可以继续阅读周志华老师的《机器学习》一书做更深入了解。“串联”形式则不断加大同一模型的复杂程度，继而通过更复杂的模型实现单一分类器表现的提升。这个思路逐渐发展成为神经网络算法，并且随着网络深度逐步提升，模型中零件由乘法 +sigmoid 激活函数，换成卷积池化 +relu 激活函数，更是进一步奠定了目前火热的深度学习算法的基石。

本章讲一讲如何用简单的代码来实现“串联”形式的计算过程。我们先来说算法，仍然基于之前逻辑回归的算法：

（1）随机初始化一组 ω，比如可以全设为 0，当然实际上不推荐这样。

（2）训练集中，逻辑回归函数里输入特征 x，计算 ωTx，得到预测结果 $\hat{y}$。

（3）计算全部训练集中逻辑回归的结果 $\hat{y}$ 和实际 y 的差别。

（4）根据上一步的差别更新 ω。

（5）重复（2）~（4）若干次（iterations）。

这里第二步、第四步有所改动。其中第二步无非是加了一层计算，比较好办，麻烦的其实是第四步，怎么更新多组数字？之前一组数字可以直接求损失函数对应参数的导数，然后乘以一个很小的学习率，减去这个数，就更新了。现在换成多组数字，怎么分别求损失函数对这些数字的导数？如果说，这里只是多了一层，直接数学推导还比较容易的话，再多几层，又应该怎么办？

仔细想一想，这个推导的过程也并非无规律可循。上一级的神经网络梯度输出会被用作下一级计算梯度的输入，同时下一级计算梯度的输出会被作为上一级神经网络的输入。于是就思考能否将这一过程抽象化，做成一个可以自动求导的框架？ OK，以 TensorFlow 为代表的一系列深度学习框架正是根据这一思路诞生的。

4.2 深度学习框架

近几年最火的深度学习框架是什么？毫无疑问，TensorFlow 高票当选。同时 Caffe、PyTorch、MXNet、CNTK 用得也非常多。这些框架虽各有优势，但都具有一些普遍特征，据 Gokula Krishnan Santhanam 总结，大部分深度学习框架都包含以下五个核心组件：

（1）张量（Tensor）的数据结构。

（2）基于张量的各种操作。

（3）计算图（Computation Graph）。

（4）自动微分（Automatic Differentiation）工具。

（5）BLAS、cuBLAS、cuDNN 等拓展包。

其中，张量（Tensor）可以理解为任意维度的数组——比如一维数组被称作向量（Vector），二维的被称作矩阵（Matrix），这些都属于张量。有了张量，就有对应的基本操作，如取某行某列的值、张量乘以常数等。运用拓展包其实就相当于使用底层计算软件加速运算。

我们今天重点介绍的就是计算图模型和自动微分两部分。首先谈谈如何实现自动求导，然后用最简单的方法实现这两部分。

4.3 基于反向传播算法的自动求导

诸如 TensorFlow 这样的深度学习框架的入门，网上有大量的几行代码、几分钟入门这样的资料，可以快速实现手写数字识别等简单任务。如果想深入了解 TensorFlow 的背后原理，可能就不是这么容易的事情了。这里简单地谈一谈这一部分。

如同逻辑回归模型的训练时，参数 ω 是随机初始化后、不断训练优化得到的一样，当我们拿到数据、训练深度神经网络时，网络中的所有参数同样也都是变量。训练模型的过程就是如何得到一组最佳变量，使预测最准确的过程。这个过程实际上就是，输入数据经过正向传播，变成预测，然后预测与实际情况的误差反向传播误差回来，更新变量。如此反复多次，得到最优的参数。这里就会遇到一个问题，神经网络这么多层，如何保证正向、反向传播都可以正确运行？

值得思考的是，这两种传播方式都具有管道传播的特征。正向传播一层一层算就可以了，上一层网络的结果作为下一层的输入。反向传播过程可以利用链式求导法则从后往前，不断将误差分摊到每一个参数的头上。我们举一个简单的例子来说明这里如何实现正向、反向传播。

首先看正向传播。给定函数$e=(a+b)(b+1)$，当 a=2、b=1 时，进行正向传播，其实就是小学乘法，即将 a=2、b=1 带入，$e=(a+b)(b+1)=3\times2=6$。反向传播过程就麻烦一些了，我们暂且可以将这个式子直接运用求导法则进行求导。

直接求解导数可能只适合一般的函数，因为大部分人对复杂函数进行求导是一件痛苦的事情。其实计算机可以借助一定的方式进行自动求导。1985 年 Rumelhart、Hinton 和 Williams 提出了反向传播算法，就基于计算图以及链式求导法则构建了复杂函数自动求导的框架。

反向传播算法分为两步：

第一步是进行正向计算，但是这里有个额外要求，就是每计算一步，都需要对该步骤各个变量的导数进行记录。如图 4-1 所示，比如计算$c=a+b$的时候，除了要计算出$c=1+2=3$，还需要求出$\frac{\partial c}{\partial a}$以及$\frac{\partial c}{\partial b}$。计算出正向的结果以及反向的导数之后，将这两组数存在计算图中供后续步骤使用。其中，正向的结果 c=3 将被用于$e=c\times d$的计算，反向的结果$\frac{\partial c}{\partial a}$以及$\frac{\partial c}{\partial b}$将被用于后续反向的计算。

第二步是反向的梯度计算。我们学微积分的时候，知道有一个反向求导法则，在这里 $e=(a+b)(b+1)$的场景下是这样的：

$$\frac{\partial e}{\partial a}=\frac{\partial e}{\partial c}\frac{\partial c}{\partial a}$$

$$\frac{\partial e}{\partial b}=\frac{\partial e}{\partial c}\frac{\partial c}{\partial b}+\frac{\partial e}{\partial d}\frac{\partial d}{\partial b}$$

图 4-1 反向传播第一步计算——保存各步骤导数（改编自 http://colah.github.io/posts/2015-08-Backprop/）

现在需要做的就是找出$\frac{\partial e}{\partial c}$以及$\frac{\partial c}{\partial a}$，找出这两个值就可以求出$\frac{\partial e}{\partial a}$。此时，这两个值在第一步正向计算时已经算出来了，然后保存在计算图里，所以这里直接将值从计算图中取出来就可以了。整个过程如图 4-2 所示。

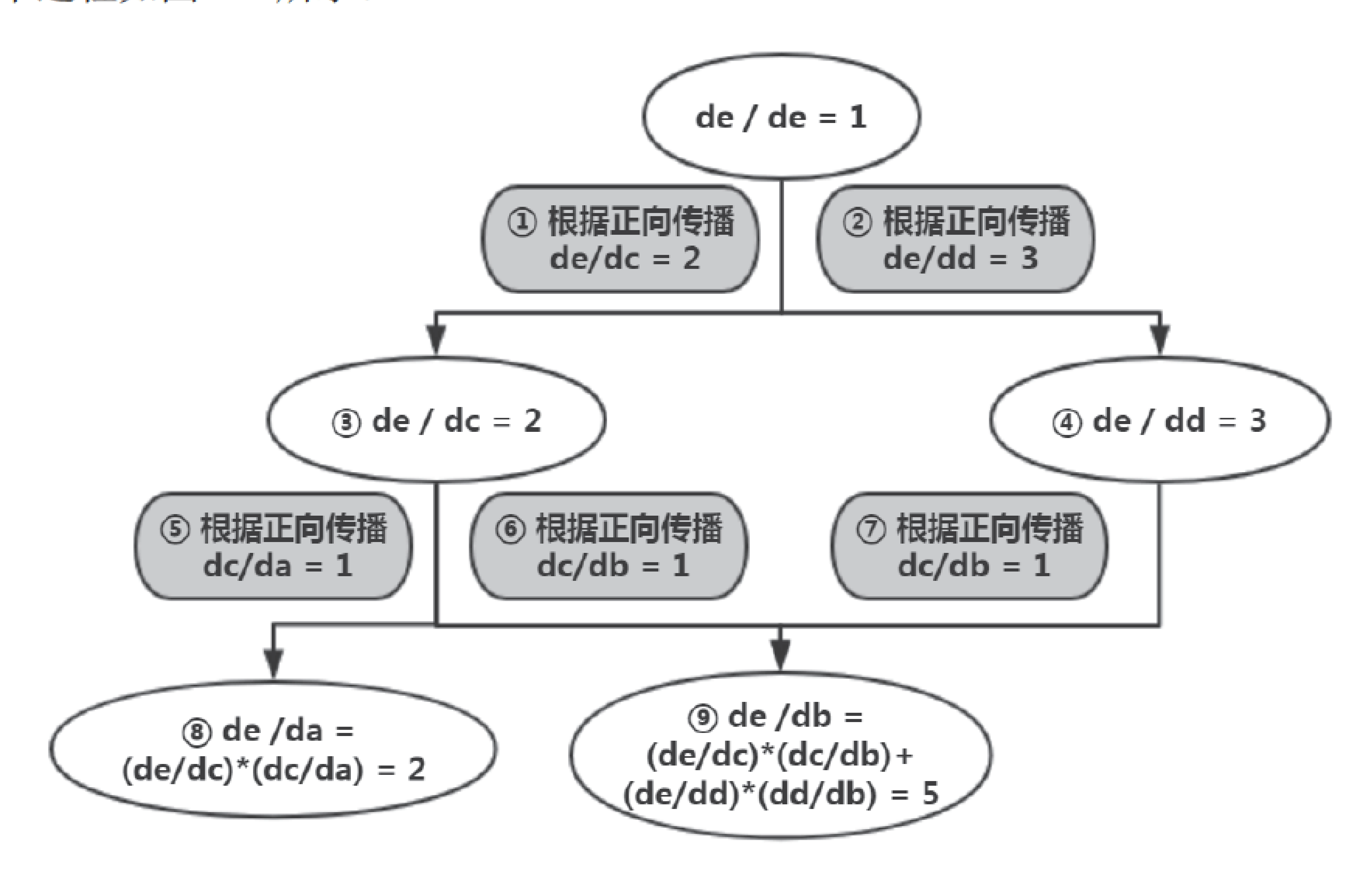


图 4-2 反向传播第二步计算——计算每一步的梯度（改编自 http://colah.github.io/posts/2015-08-Backprop/）

我们考虑如何用计算机程序表示这个过程。发现其实可以将各种计算方法（加法、乘法、逻辑回归函数等）抽象成一个对象，这个对象有两种方法，一种是正向的，即根据输入得到输出，类似输入 1+1 后输出 2；另一种则是反向的，已知当前值，输出对各个输入量的导数。然后我们用各种实际操作来继承这个对象，比如乘法操作的正向就是输入的两个数字相乘、反向就是返回另一个乘数；加法操作的正向就是两个数字相加、反向返回常数 1。

接下来，以 Torch 框架的源码为例来解释一下如何具体实现。列举 Torch 的例子，是因为 Torch 的代码文件结构比较简单，TensorFlow 的情况 Torch 比较近似，但文件结构相对更加复杂，有兴趣的可以仔细读读相关文章（http://www.cnblogs.com/yao62995/p/5773018.html）。

我们看 Torch nn 模块的源码（https://github.com/torch/nn/blob/master/）。nn 模块包括了神经网络（neural network, nn）的各种单元，包括矩阵相乘、卷积、sigmoid 函数等。我们发现几乎这个模块目录下的所有 .lua 文件都有这两个函数：

```
# lua
function xxx:updateOutput(input)
   input.THNN.xxx_updateOutput(
      input:cdata(),
      self.output:cdata()
   )
   return self.output
end

function xxx:updateGradInput(input, gradOutput)
   input.THNN.xxx_updateGradInput(
      input:cdata(),
      gradOutput:cdata(),
      self.gradInput:cdata(),
      self.output:cdata()
   )
   return self.gradInput
end
```

这里其实是相当于留了两个方法的定义，没有写具体功能。具体功能的代码在 ./lib/THNN/generic 目录（https://github.com/torch/nn/tree/master/lib/THNN/generic）中用 C 语言实现，具体以 Sigmoid 函数为例。

我们知道 Sigmoid 函数的形式是：

$$s(x) = \frac{1}{1+\mathrm{e}^{-x}}$$

代码实现起来是这样：

```
// c
void THNN_(Sigmoid_updateOutput)( THNNState
```

```
    *state, THTensor
    *input, THTensor
    *output)
  {
    THTensor_(resizeAs)(output, input);
    TH_TENSOR_APPLY2(real, output, real, input,
      *output_data = 1./(1.+ exp(- *input_data));
    );
  }
```

Sigmoid 函数求导变成：

$$s'(x) = s(x)(1 - s(x))$$

所以，这里在实现的时候就是：

```
  // c
  void THNN_(Sigmoid_updateGradInput)(
            THNNState *state,
            THTensor *input,
            THTensor *gradOutput,
            THTensor *gradInput,
            THTensor *output)
  {
    THNN_CHECK_NELEMENT(input, gradOutput);
    THTensor_(resizeAs)(gradInput, output);
    TH_TENSOR_APPLY3(real, gradInput, real, gradOutput, real, output,
      real z = * output_data;
      *gradInput_data = *gradOutput_data * (1. - z) * z;
    );
  }
```

注意，在 updateOutput 函数中，output_data 在等号左边，input_data 在等号右边；在 updateGradInput 函数中，gradInput_data 在等号左边，gradOutput_data 在等号右边。这里，output = f(input) 对应的是正向传播；input = f(output) 对应的是反向传播。

4.4 简单的深度神经网络框架实现

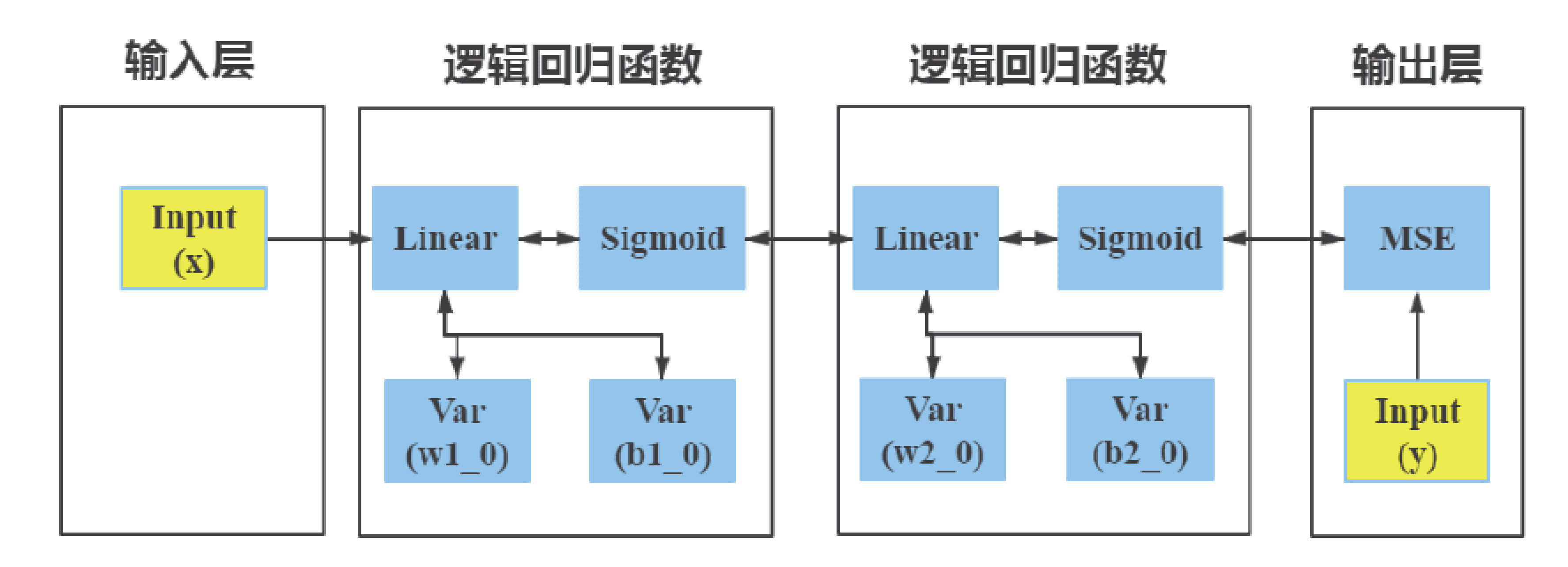
我们谈谈如何实现一个类似 TensorFlow 的计算框架，就是说可以跳过这一部分、直接使用现成的框架，如果现有框架缺乏一些新的内容时，一定的代码实现能力就会显得比较重要。因此，这里借用网上的一个小型开源框架 Miniflow 代码（https://github.com/BillZito/miniflow）中的核心内容来简单谈谈如何实现深度神经网络框架，继而改造第 2 章中的鸢尾花数据集的逻辑回归算法的代码。这个小型开源项目同样来自优达学城。

上一节提到过，深度神经网络需要实现五大核心组件，Miniflow 的程序选择了相对简单而又好实现的计算图并且重写了自动微分部分，如表 4-1 所示。

表4-1　五大核心组件的实现方法

核心组件	实现方法
张量（Tensor）	使用numpy库
基于张量的各种操作	使用numpy库
BLAS、cuBLAS、cuDNN等拓展包	借助numpy库使用BLAS，不使用GPU
计算图（Computation Graph）	代码通过Kahn算法实现
自动微分（Automatic Differentiation）工具	代码通过链式求导法则实现

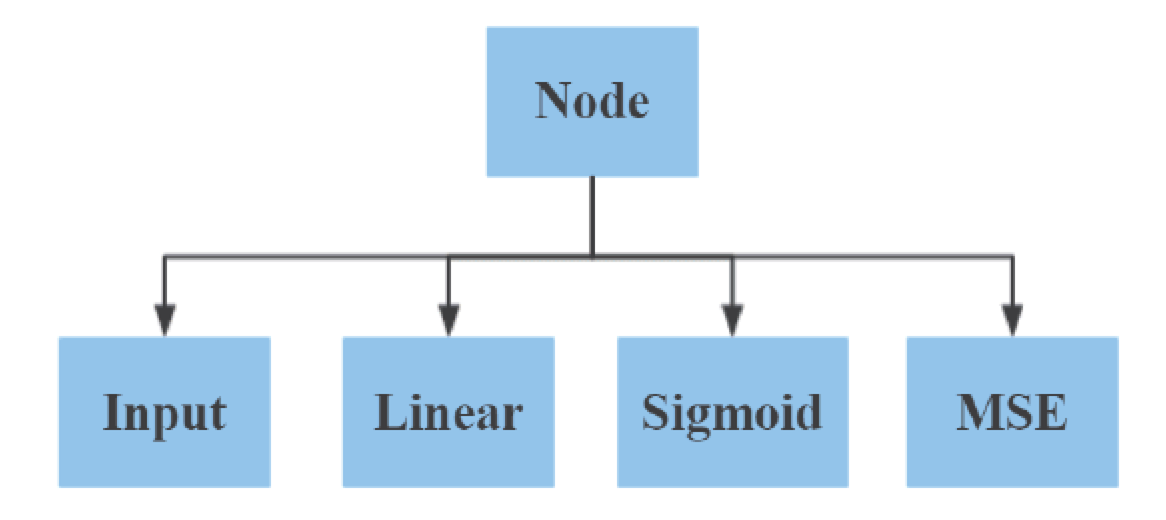
接下来的部分将用 Miniflow 框架搭建这样的一个神经网络：



重新分析第 2 章逻辑回归鸢尾花分类的数据。

4.4.1　数据结构部分

首先实现一个父类 Node，然后基于这个父类依次实现 Input Linear Sigmoid 等模块。这里运用了简单的 Python Class 继承。



具体而言，首先写一个 Node 类，这个类有 forward backward 两种方法，但是这两种方法先空着不写。我们基于 Node 类实现 Input Linear 这些模块时，需要将 forward 和 backward 两种方法针对每个模块分别重写。

```
import numpy as np
import matplotlib.pyplot as plt

%matplotlib inline
```

```
# python
class Node(object):
    def __init__(self, inbound_nodes=[]):
        self.inbound_nodes = inbound_nodes
        self.value = None
        self.outbound_nodes = []

        self.gradients = {}

        for node in inbound_nodes:
            node.outbound_nodes.append(self)

    def forward(self):
        raise NotImplementedError

    def backward(self):
        raise NotImplementedError

class Input(Node):
    def __init__(self):
        Node.__init__(self)

    def forward(self):
        pass

    def backward(self):
        self.gradients = {self: 0}
        for n in self.outbound_nodes:
            self.gradients[self] += n.gradients[self]

class Linear(Node):
    def __init__(self, X, W, b):
        Node.__init__(self, [X, W, b])

    def forward(self):
        X = self.inbound_nodes[0].value
        W = self.inbound_nodes[1].value
        b = self.inbound_nodes[2].value
        self.value = np.dot(X, W) + b

    def backward(self):
        self.gradients = {n: np.zeros_like(n.value) for n in self.
inbound_nodes}
        for n in self.outbound_nodes:
            grad_cost = n.gradients[self]
            self.gradients[self.inbound_nodes[0]] += np.dot(grad_cost,
self.inbound_nodes[1].value.T)
```

```
                self.gradients[self.inbound_nodes[1]] += np.dot(self.
inbound_nodes[0].value.T, grad_cost)
                self.gradients[self.inbound_nodes[2]] += np.sum(grad_cost,
axis=0, keepdims=False)

    class Sigmoid(Node):
        def __init__(self, node):
            Node.__init__(self, [node])

        def _sigmoid(self, x):
            return 1. / (1. + np.exp(-x))

        def forward(self):
            input_value = self.inbound_nodes[0].value
            self.value = self._sigmoid(input_value)

        def backward(self):
            self.gradients = {n: np.zeros_like(n.value) for n in self.
inbound_nodes}
            for n in self.outbound_nodes:
                grad_cost = n.gradients[self]
                sigmoid = self.value
                self.gradients[self.inbound_nodes[0]] += sigmoid * (1 -
sigmoid) * grad_cost

    class MSE(Node):
        def __init__(self, y, a):
            Node.__init__(self, [y, a])

        def forward(self):
            y = self.inbound_nodes[0].value.reshape(-1, 1)
            a = self.inbound_nodes[1].value.reshape(-1, 1)

            self.m = self.inbound_nodes[0].value.shape[0]
            self.diff = y - a
            self.value = np.mean(self.diff**2)

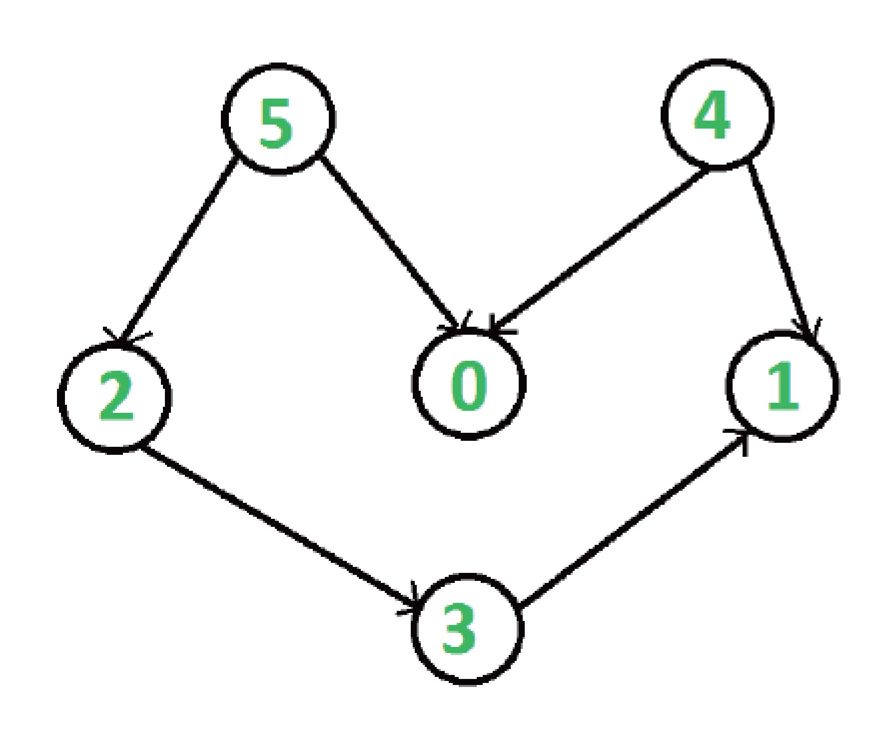
        def backward(self):
            self.gradients[self.inbound_nodes[0]] = (2 / self.m) * self.diff
            self.gradients[self.inbound_nodes[1]] = (-2 / self.m) * self.diff
```

4.4.2 计算图部分

接下来定义计算图。这里定义计算图，需要做到两点，首先是计算图中各个节点中需要保存哪些参数；其次是这些节点在计算的过程中采用什么样的计算顺序。比如之前的例子中，要计算 e 就要先把（a+b）以及（b+1）的结果分别算出来，以及这些结果分别依赖哪些东西。

多说一点，TensorFlow 使用了静态图，对 C 程序编译比较熟悉的读者应该意识到了，这里其实很类似编写 Makefile，不同的是通过 Makefile 定义了子程序之间的相互依赖关系，编译某个子程序之前，需要完成这个子程序需要依赖的所有程序的编译。实际上并非所有的深度学习框架都是这样的，Torch 的执行过程就是动态的，TensorFlow 基于静态图，这与 TensorFlow 的作者 Jeff Dean 精通编译原理有很大的关系。

我们这里在定义计算图各个节点之间的计算顺序时使用 Kahn 算法作为调度方法，就是首先 TensorFlow 的各个模块会生成一个有向无环图，每个模块作为一个节点，模块之间相互依赖关系作为图的边，如图 4-3 所示。



（图片来源：http://www.geeksforgeeks.org/topological-sorting/）

图 4-3 计算图中各个节点之间存在一定的依赖顺序

在计算过程中，几个模块存在着相互依赖关系，比如要计算模块 1，就必须完成模块 3 和模块 4，而要完成模块 3，就需要在之前按顺序完成模块 5、2；这里可以使用 Kahn 算法作为调度算法（下面的 topological_sort 函数），从计算图中推导出类似 5->2->3->4->1 的计算顺序。

```
# python
def topological_sort(feed_dict):
    input_nodes = [n for n in feed_dict.keys()]
    G = {}
    nodes = [n for n in input_nodes]
    while len(nodes) > 0:
        n = nodes.pop(0)
        if n not in G:
            G[n] = {'in': set(), 'out': set()}
        for m in n.outbound_nodes:
            if m not in G:
                G[m] = {'in': set(), 'out': set()}
            G[n]['out'].add(m)
            G[m]['in'].add(n)
            nodes.append(m)

    L = []
```

```
    S = set(input_nodes)
    while len(S) > 0:
        n = S.pop()
        if isinstance(n, Input):
            n.value = feed_dict[n]

        L.append(n)
        for m in n.outbound_nodes:
            G[n]['out'].remove(m)
            G[m]['in'].remove(n)
            if len(G[m]['in']) == 0:
                S.add(m)
    return L
```

4.4.3 使用方法

定义完图的模块以及执行顺序的调度方式之后，我们开始逐个模块地进行正向计算、反向求导：

```
def forward_and_backward(graph):
    for n in graph:
        n.forward()

    for n in graph[::-1]:
        n.backward()
```

首先，由图的定义生成执行顺序：

```
graph = topological_sort(feed_dict)
```

其次，对各个模块顺次执行正向计算反向求导：

```
forward_and_backward(graph)
```

最后，简单介绍一下随机梯度下降。随机梯度下降将在第 7 章详细讲解，这里只是提前使用，让读者感受一下。其核心思想是，在第 2 章中用的是所有 150 个样本的数据，训练逻辑回归模型，通过计算导数作为梯度乘以学习率，更新参数ω ，然后多次重复这个过程。这里升级成使用随机梯度下降的话就不再使用全部样本，而是每次从 150 个样本中随机抽样若干，代表所有样本来训练参数。

```
def sgd_update(trainables, learning_rate=1e-2):
    for t in trainables:
        t.value = t.value - learning_rate * t.gradients[t]
```

接下来使用这个模型。同样使用第 2 章的鸢尾花数据集：

```
from sklearn.utils import resample
from sklearn import datasets

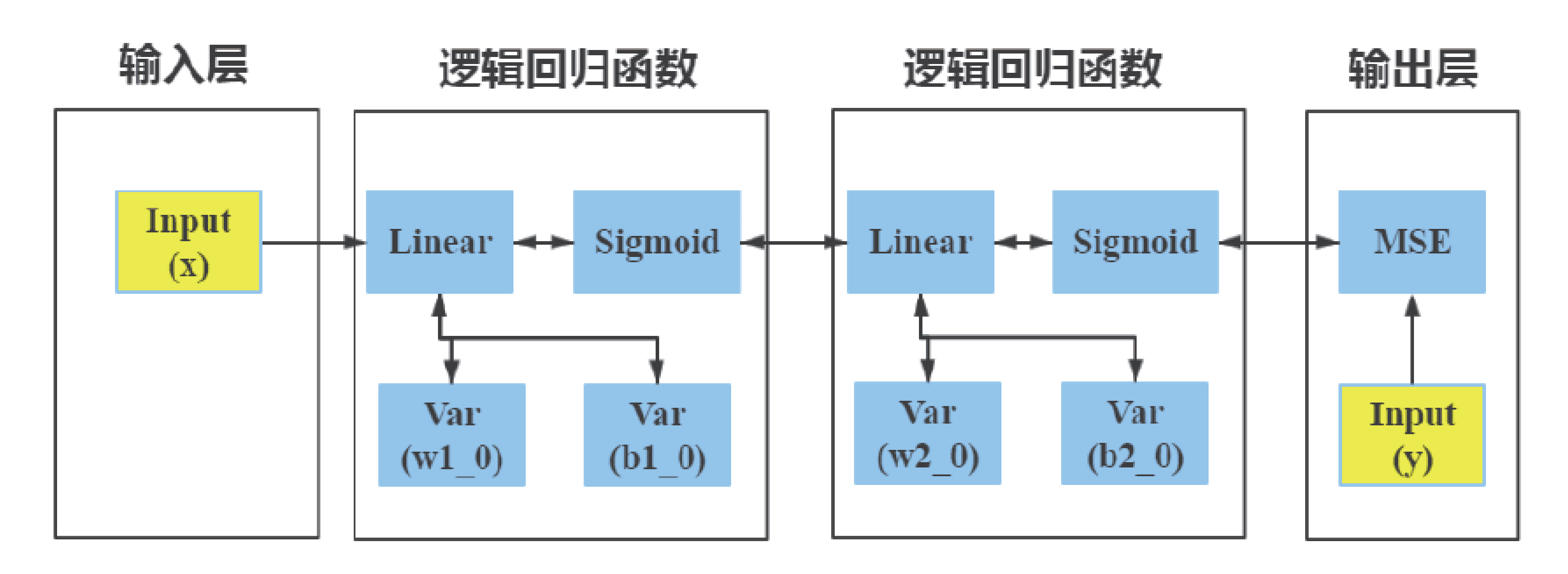
%matplotlib inline

data = datasets.load_iris()
X_ = data.data
y_ = data.target
y_[y_==2] = 1 # 0 for virginica, 1 for not virginica
print(X_.shape, y_.shape)
```

运行结果：

```
# out:
(150,4), (150,)
```

我们根据前文提到的“串联逻辑回归”的图纸，用写的模块来定义这个神经网络：




```
np.random.seed(0)
n_features = X_.shape[1]
n_class = 1
n_hidden = 3

X, y = Input(), Input()
W1, b1 = Input(), Input()
W2, b2 = Input(), Input()

l1 = Linear(X, W1, b1)
s1 = Sigmoid(l1)
l2 = Linear(s1, W2, b2)
t1 = Sigmoid(l2)
cost = MSE(y, t1)
```

4.4.4 训练模型

接下来初始化参数值，定义训练参数进行训练：

```
    # 随机初始化参数值
    W1_0 = np.random.random(X_.shape[1]*n_hidden).reshape([X_.shape[1], n_
hidden])
    W2_0 = np.random.random(n_hidden*n_class).reshape([n_hidden, n_class])
    b1_0 = np.random.random(n_hidden)
    b2_0 = np.random.random(n_class)

    # 将输入值带入算子
    feed_dict = {
        X: X_,    y: y_,
        W1: W1_0, b1: b1_0,
        W2: W2_0, b2: b2_0
    }

    # 训练参数
    # 这里训练 100 轮（eprochs），每轮抽 4 个样本（batch_size）训练 150/4 次（steps_
per_eproch），学习率 0.1
    epochs = 100
    m = X_.shape[0]
    batch_size = 4
    steps_per_epoch = m // batch_size
    lr = 0.1

    graph = topological_sort(feed_dict)
    trainables = [W1, b1, W2, b2]

    l_Mat_W1 = [W1_0]
    l_Mat_W2 = [W2_0]

    l_loss = []
    for i in range(epochs):
        loss = 0
        for j in range(steps_per_epoch):
            X_batch, y_batch = resample(X_, y_, n_samples=batch_size)
            X.value = X_batch
            y.value = y_batch

            forward_and_backward(graph)
            sgd_update(trainables, lr)
            loss += graph[-1].value

        l_loss.append(loss)
        if i % 10 == 9:
            print("Eproch %d, Loss = %1.5f" % (i, loss))
```

运行结果：

```
# out:
Eproch 9, Loss = 7.76529
Eproch 19, Loss = 8.26954
Eproch 29, Loss = 7.45415
…
Eproch 79, Loss = 7.32178
Eproch 89, Loss = 3.70127
Eproch 99, Loss = 1.19249
```

模型的 Loss 误差值逐步减小，说明模型预测的结果与真实情况的误差也在逐步减少。可以画一下整体情况，如图 4-4 所示。

```
plt.plot(l_loss)
plt.title("Cross Entropy value")
plt.xlabel("Eproch")
plt.xlabel("Loss")
```

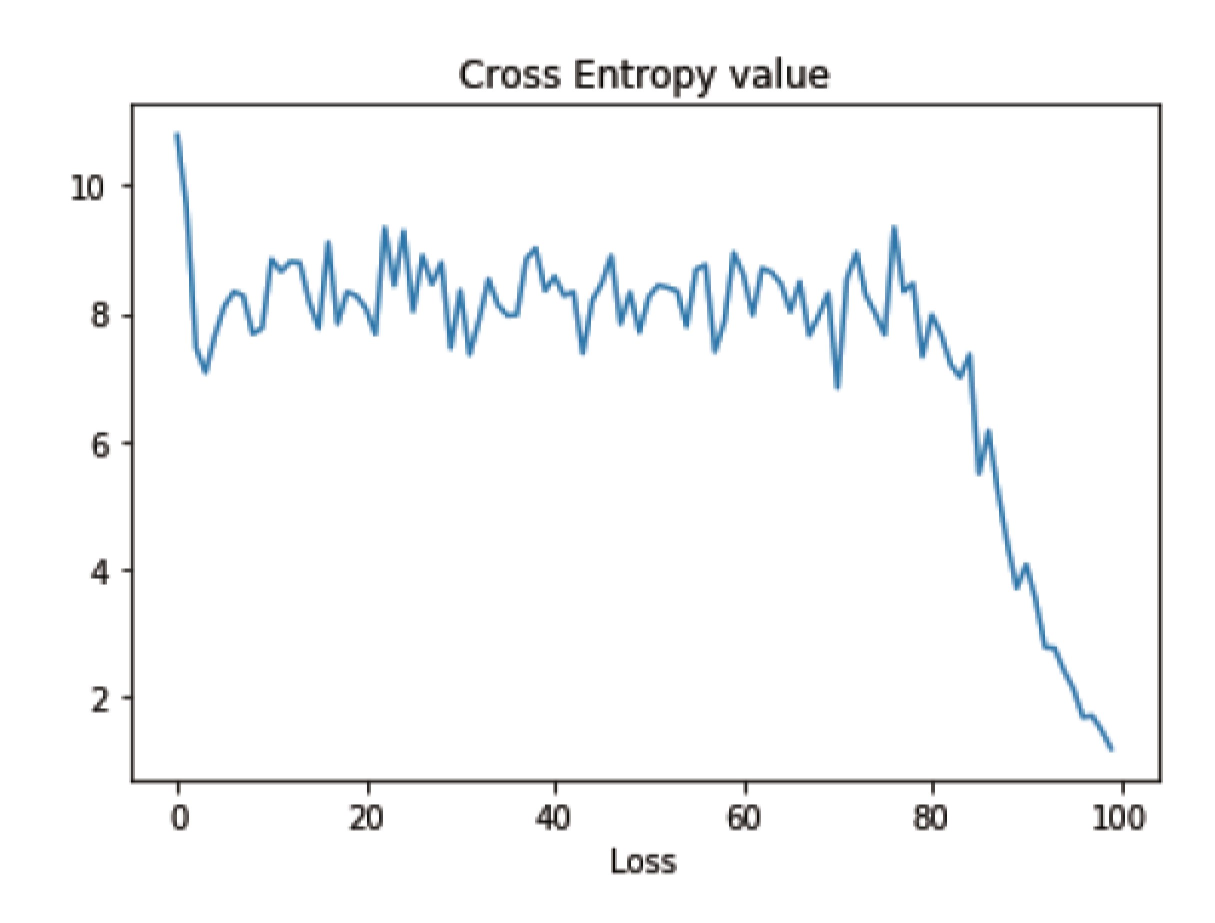


图 4-4 模型预测结果与真实情况误差（y 轴）随着训练次数（x 轴）增加逐渐减小

最后用模型预测所有数据的情况，如图 4-5 所示。

```
X.value = X_
y.value = y_
for n in graph:
    n.forward()

plt.plot(graph[-2].value.ravel())
plt.title("Predict for all 150 Iris data")
plt.xlabel("Sample ID")
plt.ylabel("Probability for not a virginica")
```

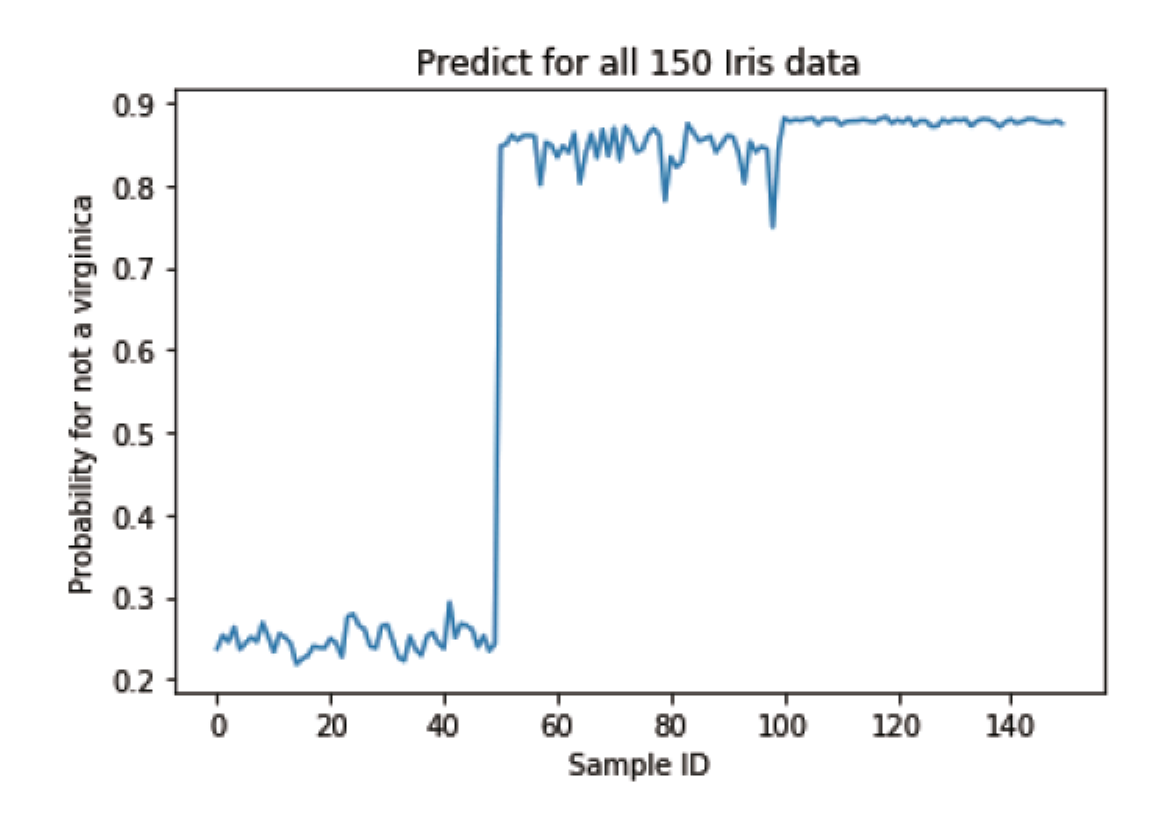


图 4-5　150 个鸢尾花样本（x 轴）是否是 virginica 分类结果的概率

得到了与第 2 章类似的结果，即后 100 个样本不是 virginica 的概率很高，与实际情况相符。

4.5　参考文献及网页链接

［1］ BillZito. BillZito/miniflow. GitHub (2017). Available at: https://github.com/BillZito/miniflow.

［2］ Calculus on Computational Graphs: Backpropagation. Calculus on Computational Graphs: Backpropagation -- colah’s blog. Available at: http://colah.github.io/posts/2015-08-Backprop/.

［3］ Rumelhart, D. E., Hinton, G. E. & Williams, R. J. Learning representations by back-propagating errors. Nature News (1986). Available at: http://www.nature.com/nature/journal/v323/n6088/abs/323533a0.html?foxtrotcallback=true.

［4］ Santhanam, G. K. The Anatomy of Deep Learning Frameworks – Gokula Krishnan Santhanam – Medium. Medium (2017). Available at: https://medium.com/@gokul_uf/the-anatomy-of-deep-learning-frameworks-46e2a7af5e47.

［5］ Topological Sorting. GeeksforGeeks (2017). Available at: http://www.geeksforgeeks.org/topological-sorting/.

［6］ yj0306. [tensorflow 源码分析] Conv2d 卷积运算（前向计算，反向梯度计算）. yj0306. 博客园 . Available at: http://www.cnblogs.com/yao62995/p/5773018.html.

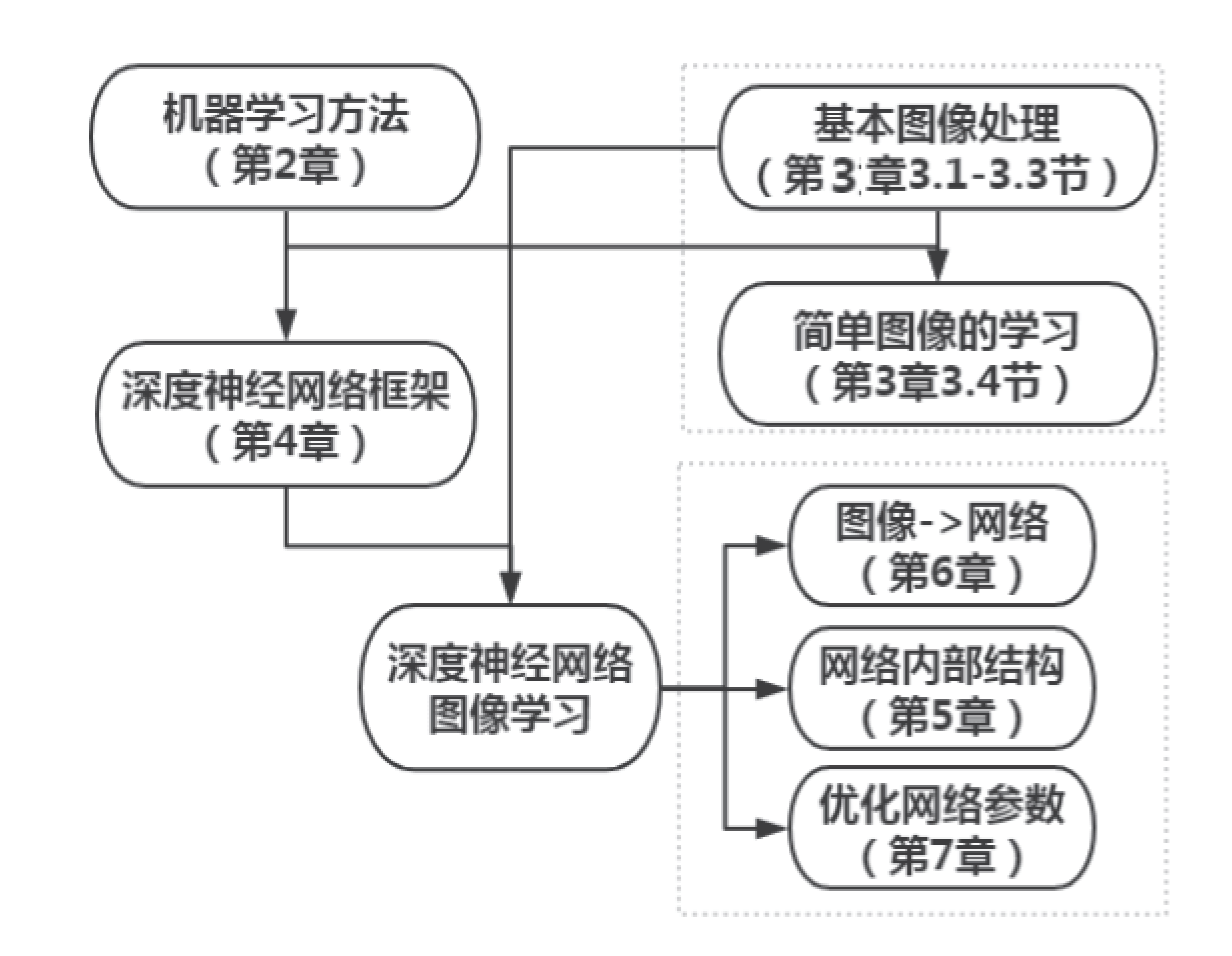
［7］周志华 . 机器学习 [M]. 北京：清华大学出版社，2016.

第 5 章

排列组合——深度神经网络框架的模型元件

从本章开始，我们正式介绍如何使用深度学习框架来进行图像的处理。

首先回顾一下前几章的内容。我们在第 2 章介绍了机器学习的一些基本概念，包括数据集的划分、数据的挖掘、模型的训练以及如何评价训练模型的准确性。然后第 3 章介绍了图像处理的基本方法，包括基于颜色、基于形态的特征处理，接下来进一步介绍如何使用机器学习方法处理这些特征。接着，第 4 章先从第 2 章的逻辑回归引申出深度学习框架的设计思想，实现了简单的神经网络算法。



我们可以进一步地将深度神经网络框架，结合基本的图像处理，利用深度神经网络，对图像进行分类、分割等处理。在这个过程中，我们可以将处理过程概括成一个三段论：

- 处理的内容**是什么**——如何将图像传入深度神经网络
- **为什么**可以得到这样的结果——使用的是什么样的神经网络结构
- 训练过程应该**怎么做**——网络参数的优化

我们在第 4 章设计了 Input Linear Sigmoid MSE 几个元件，组装出一个两层的神经网络。那么能否将上一章的简单神经网络进一步地升级成深度神经网络呢？当然可以，此时需要更多的零件，然后组装成为更深的网络。这些零件可以大致概括为：

- 常用层
  - Dense
  - Activation
  - Dropout
  - Flatten
- 卷积层
  - Conv2D
  - Cropping2D
  - ZeroPadding2D
- 池化层
  - MaxPooling2D
  - AveragePooling2D
  - GlobalAveragePooling2D
- 正则化层
  - BatchNormalization

- 反卷积层（Keras 中在卷积层部分）
  - UpSampling2D
  - 循环层
  - SimpleRNN
  - LSTM
  - GRU

需要强调一下，这些层与之前一样，都同时包括了正向传播、反向传播两条通路。我们这里只介绍比较好理解的正向传播过程，基于其导数的反向过程同样也是存在的，其代码已经包括在 TensorFlow 的框架中对应的模块里，可以直接使用。

当然，还有更多的零件，具体可以去 Keras 文档 http://keras-cn.readthedocs.io/en/latest/ 中参阅。我们这里选一些常用的进行简单的介绍。

5.1 常用层

5.1.1 Dense

全连接层（Dense）就是第 4 章中提到的 Linear 层，即 $y = \omega x + b$，计算乘法以及加法。由于矩阵乘法的特点，我们可以通过乘法操作，连接所有输入点以及输出点，因此也被称作全连接层。

5.1.2 Activation

激活层（Activation）在第 4 章中同样出现过，即 sigmoid 层。当然，激活层不止有 sigmoid 这一种形式，比如有 thanh、ReLU、softplus，如图 5-1 所示。

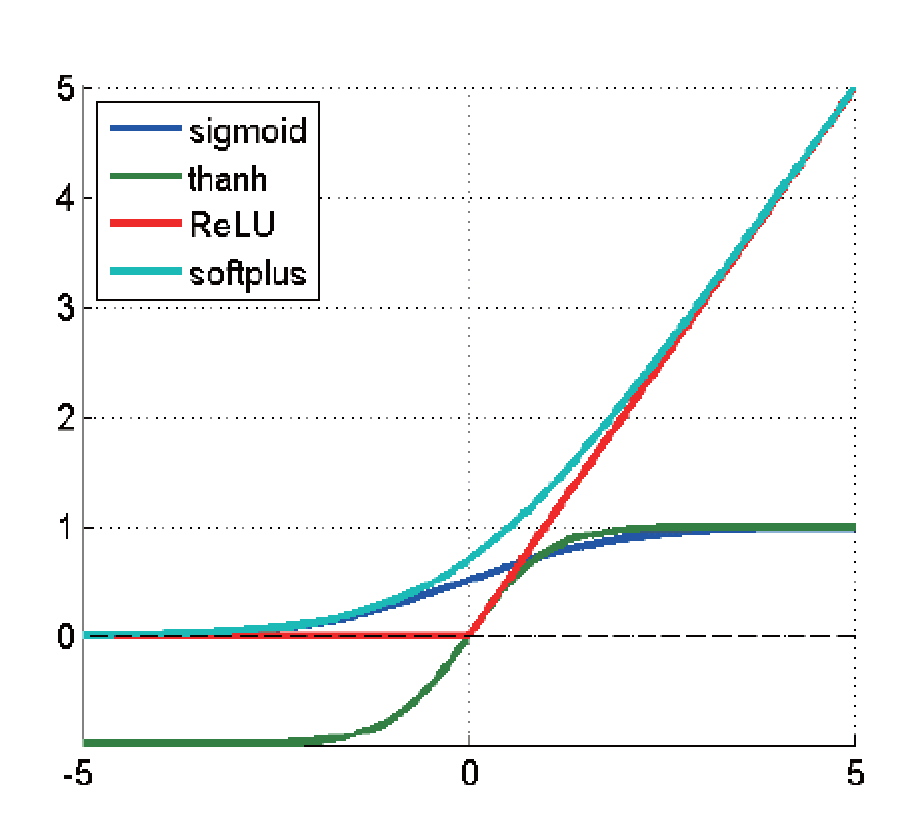


图 5-1 常见的激活层函数（图片来源：http://m.blog.csdn.net/article/details?id=52890463）

其中，ReLU 层可能是深度学习时代最重要的一种激发函数，在 2011 年首次被提出。由公式可见，相比于早期的 thanh 与 sigmoid 函数，ReLU 有两个重要的特点：其一是在较小处有一个下限 0，但是较大值 ReLU 函数没有取值上限；其二是 ReLU 层在 0 处不可导，是一个非线性的函数，

$$relu(x) = \begin{cases} y = x, x > 0 \\ y = 0, x \leqslant 0 \end{cases}$$

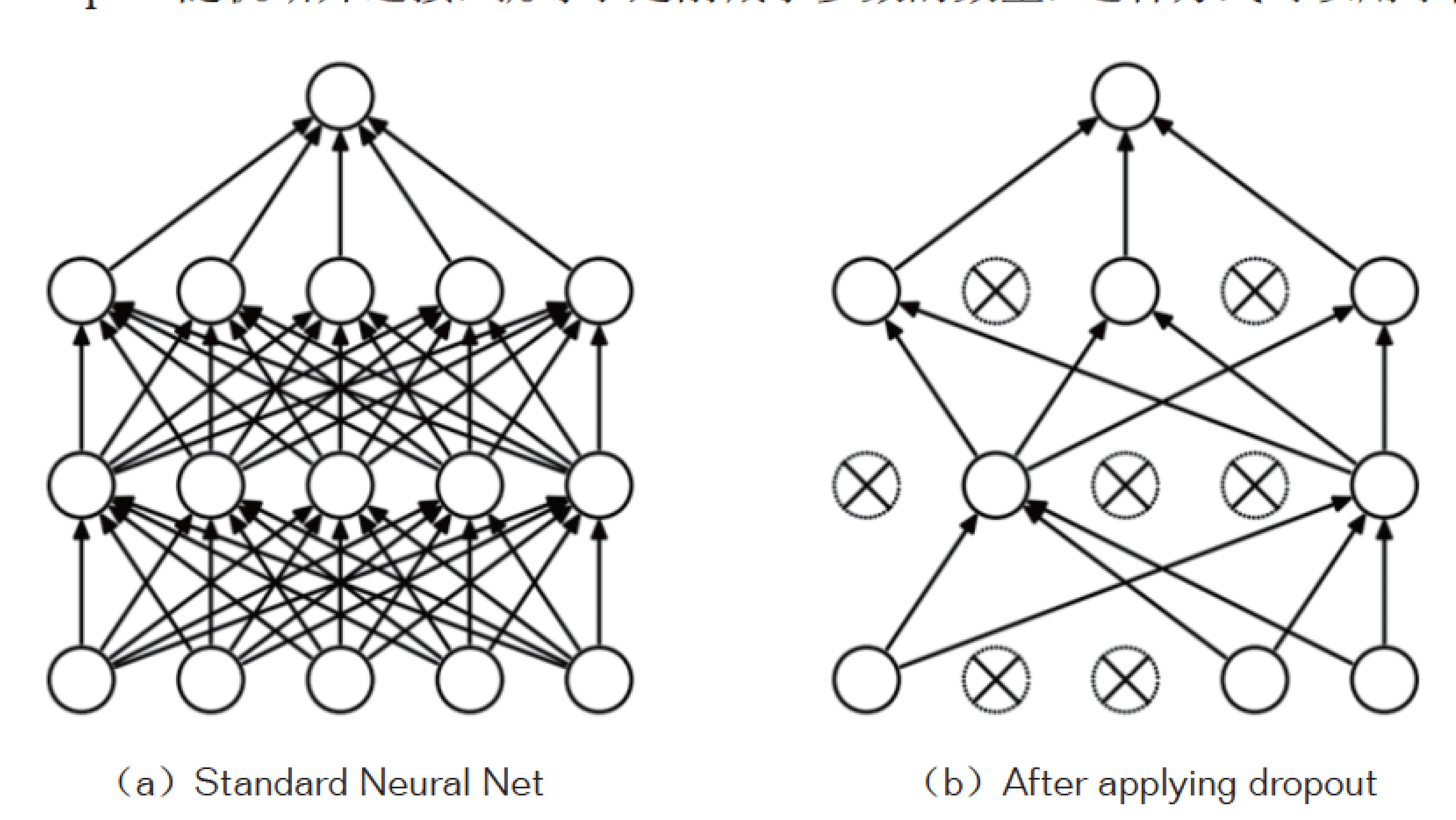
即$y = x *$（$x > 0$）时对其求导，其结果是：

$$\frac{\partial relu(x)}{\partial x} = \begin{cases} y = 1, x > 0 \\ y = 0, x < 0 \end{cases}$$

函数本身取值没有上限、求导后大于零时导数固定，相比之前的激活函数，在深度神经网络中，这是一个非常重要的优点——sigmoid 激发层的导数在数值的绝对值很高时是接近 0 的，只有在大概 [-5, 5] 的区间内才会取一个较大的值，这就会导致多层神经网络在传递误差时，在 sigmoid 层乘以一个很小的数，导致误差的**梯度消失**。而如果换成 ReLU 层，传入的导数在其大于 0 的状态下会直接返回输入误差，这样误差的梯度就可以在各层中得到保留，从而实现多层神经网中误差从底层向顶层的有效传递。

5.1.3 Dropout

失活（Dropout）层（见图 5-2）指的是在训练过程中，每次更新参数时将会随机断开一定百分比（rate）的输入神经元。这种方式与第 2 章提到的正则化有相似支持——正则化会给所有参数乘以一个系数，共同计算损失函数，为了避免损失函数过高，模型参数的数量、数值都会缩小。而这里使用 Dropout 随机断开连接，就等于是削减了参数的数量，这种方式可以用于防止过拟合。



（图片来源：Dropout: A Simple Way to Prevent Neural Networks from Overfitting）

图 5-2　失活层的基本原理

5.1.4 Flatten

展开层（Flatten）指的是将高维的张量（Tensor，如二维的矩阵、三维的 3D 矩阵等）变成一个一维张量（向量）。Flatten 层通常位于连接深度神经网络的卷积层部分以及全连接层部分。

5.2 卷积层

我们在第 2 章的结尾提到：

传统的机器学习模型，缺乏特征组合能力，尤其是对图像输入，计算机可以理解单独的一个像素，但是把单一像素，同周围三五个点一起考虑，计算机模型在组合的时候，似乎不太能把握这一组点的关系。

使用卷积层的目的，就是组合周围多个像素、进一步提取特征，再配合池化进一步整合。这一“局部相近像素点具有相近含义、可以被组合”的观点，在《深度学习》中被描述成一种“无限强的先验”。

5.2.1 Conv2D

这里以 2D 的卷积神经网络为例来逐一介绍卷积神经网络中的重要函数。比如使用一个形状如下的卷积核：

1	0	1
0	1	0
1	0	1

扫描这样一个二维矩阵：

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

如果定义卷积的步长为 1，不扫描边缘区域，则将进行 9 次卷积计算，其中第 1、2、3、9 次卷积操作的过程（左）及卷积的结果（右）如图 5-3 所示。

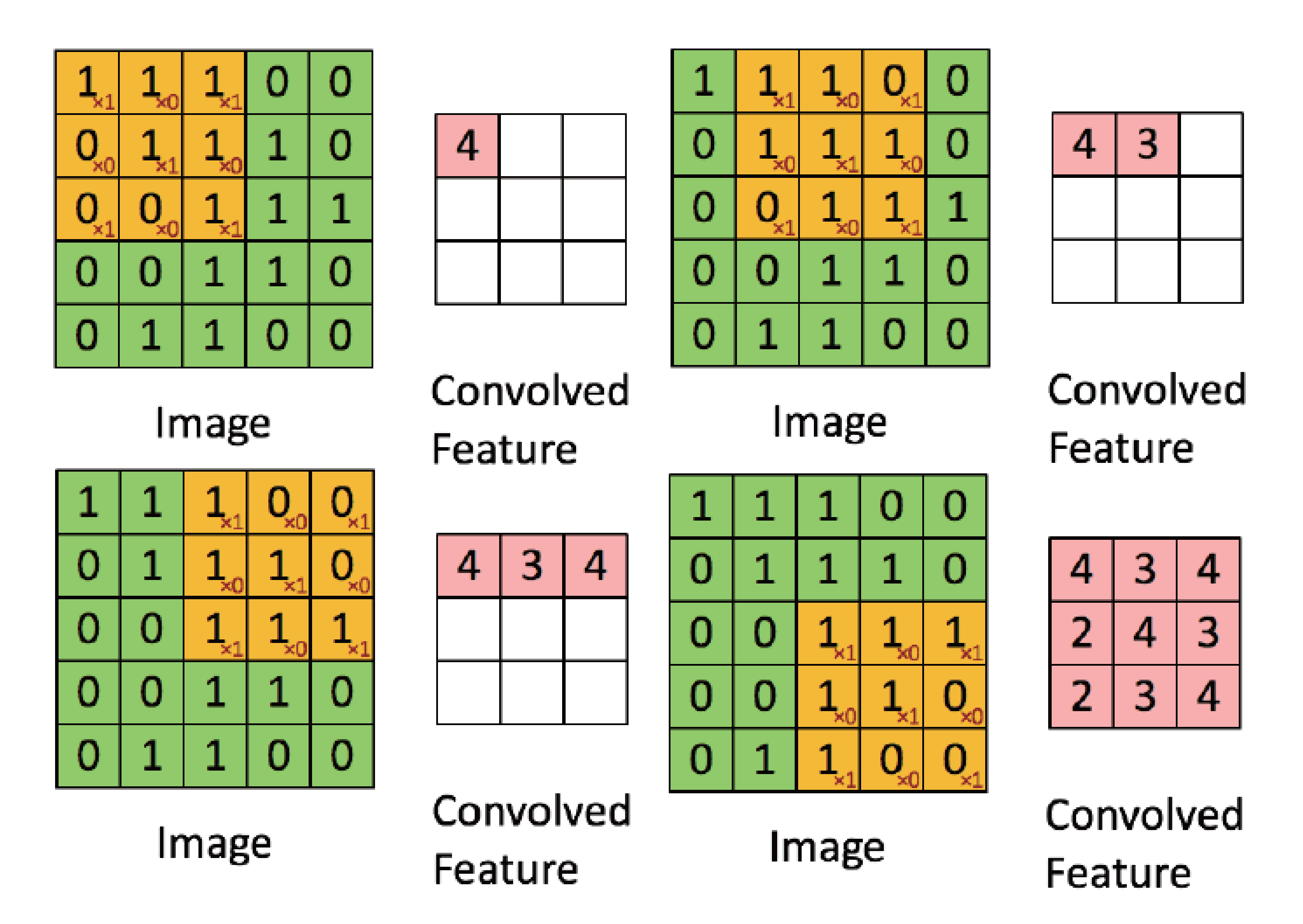


图 5-3　卷积层扫描图像的过程（图片来源：kdnuggets）

由于 Keras 的 Conv2d 函数不支持指定卷积核，我们用 TensorFlow 底层代码来重复一下图 5-3 中的卷积过程：

```
import tensorflow as tf
import numpy as np

x_input = np.array([
        [1,1,1,0,0],
        [0,1,1,1,0],
        [0,0,1,1,1],
        [0,0,1,1,0],
        [0,1,1,0,0]
], dtype=np.float32)

x_kernel_1 = np.array([
    [1,0,1],
    [0,1,0],
    [1,0,1]
], dtype=np.float32)

tf_x_input  = tf.constant(np.reshape(x_input, newshape=[1,5,5,1]))
tf_x_kernel_1 = tf.constant(np.reshape(x_kernel_1, newshape=[3,3,1,1]))

y1 = tf.nn.conv2d(tf_x_input, tf_x_kernel_1, strides=[1,1,1,1],
padding="VALID")
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    [y1_cov] = sess.run([y1])

y1_cov[0,:,:,0]
```

运行结果：

```
# out:
array([[ 4.,  3.,  4.],
       [ 2.,  4.,  3.],
       [ 2.,  3.,  4.]], dtype=float32)
```

如果使用 Keras， 其输入参数定义如下，可以看到 Keras 可以指定卷积核的大小、层数，但没有像底层函数 tf.nn.conv2d 一样提供指定卷积核的接口：

```
Conv2D(filters, kernel_size, strides=(1, 1), padding='valid', ...)
```

其中：

- filters 指的是输出的卷积层的层数。通常看见的介绍资料都只显示输出了一个卷积层，即 filters =1，而实际运用过程中卷积层会接受多层输入，然后返回的结果也是多层输出。接下来将仔细讨论这部分内容（见下文错误 2）。
- kernel_size 指的是卷积层的大小，是一个二维数组，分别代表卷积层有几行、几列。
- strides 指的是卷积核在输入层扫描时，在 x,y 两个方向每间隔多长执行一次扫描。
- padding 指的是是否扫描边缘。如果是 valid，则仅仅扫描已知的矩阵，即忽略边缘。如果是 same，则将根据情况在边缘补上 0，并且扫描边缘，使得输出的大小等于 input_size / strides。

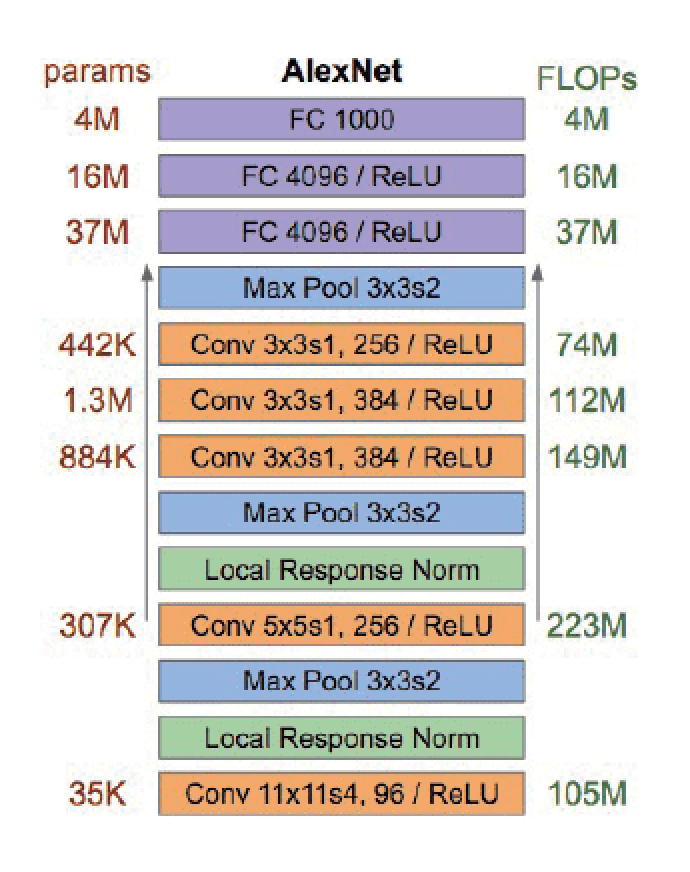
关于卷积的参数，有几个常见的错误观点：

- 错误 1：深度神经网络中，卷积核的权重是固定的（如上面 3×3 卷积核中对角线部分是 1，其他是 0，又或者传统图像处理中使用的高斯分布、Sobel 算子等经典卷积核）。
- 错误 2：假如卷积层有 m 个输入维度，如图片的 RGB 图层中 m ＝ 3，用 n 个卷积层扫描输入图层，则输出层一共是 m×n。
- 错误 3：可以像训练逻辑回归一样，随便初始化一个卷积核的权重以后，扔给模型让它调整就好。

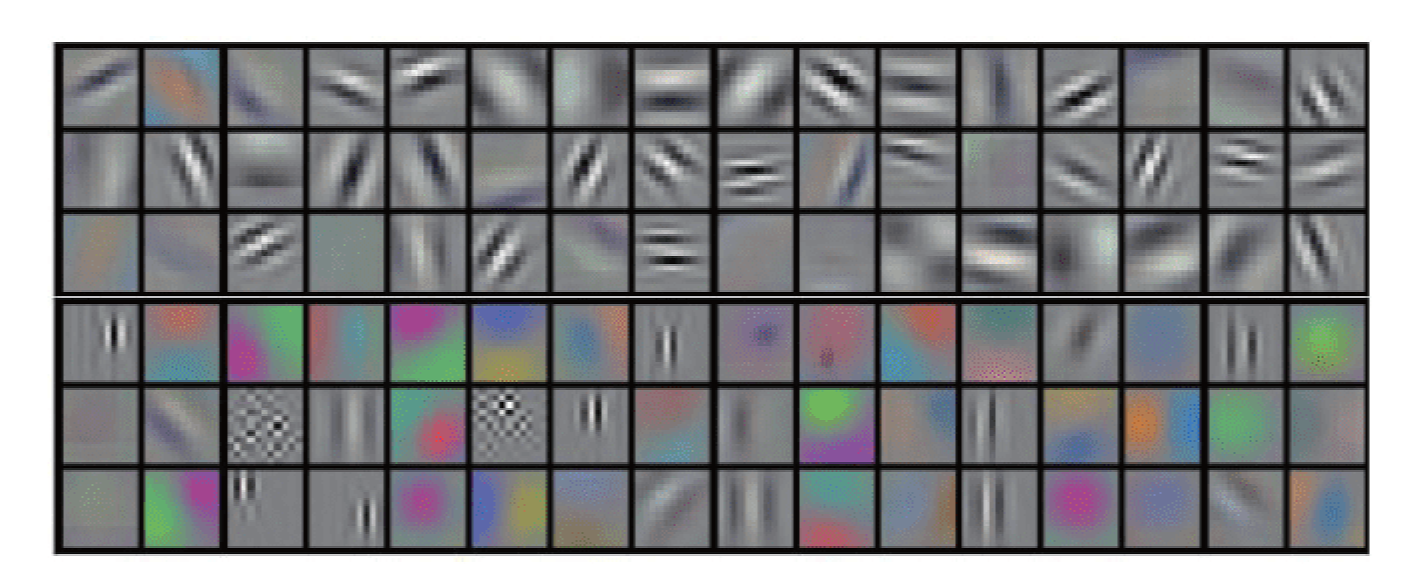
1. 关于卷积核的第一种错误认识——权值固定

对于错误 1，最直接的解释是，如果卷积核固定，为什么 Keras 不提供让我们指定卷积核的接口？可以查看一个卷积神经网络的实战案例，即著名的 AlexNet。首先这里有 5 层卷积核，每层卷积核的个数（96 256 384 384 256）又各不相同，如果这么多卷积核的参数都需要数据分析人员钦定（如本书 3.3 节中用的 Sobel 算子），然后模型表现又不好，这些固定下来的卷积核里面的参数应该如何改进呢？所以深度神经网络的训练过程中，这些值都是变化的，让模型自己去找最优解（如图 5-4 中的右图）。由于是卷积核变化的，高级封装的 Keras 就不再提供卷积核的直接输入接口了，只有 TensorFlow 的底层 nn.conv2d 函数才提供。

AlexNet 结构



第一层卷积层的 96 个、大小为 11×11 的卷积核结果矩阵



（图片来源：AlexNet 2012 年文章 ImageNet Classification with Deep Convolutional Neural Networks）

图 5-4　AlexNet 结构

2. 关于卷积核的第二种错误认识——卷积只做扫描

对于错误 2，同样以 AlexNet 的工程实践为例来谈谈后果。这里如果 AlexNet 输入的 RGB 图像三层，第一次卷积乘以 96 输出 288 图层，第二次卷积 288 再乘以 256，然后一路乘下去，整个模型输出的图层数量很快就会“爆炸”。实际上在扫描出例如第一层的 3×96=288 个输出后，会有一个 288 到 96 的全连接，即深度神经网络中的卷积层，实际上是一个计算卷积后再对卷积结果进行全连接，卷积层的这种特点称为**稀疏权重**。

稀疏权重的意思是，对同样大小的输入输出，如果直接全连接，连接次数就是输入矩阵中的点数乘以输出矩阵中的点数。以 AlexNet 第一层为例，这个值是 (224×224×3)×(55×55×96)，连接数是巨大的，但是引入卷积核作为媒介以后，等于是距离较近的点通过卷积局部连接一次，然后较远的点通过卷积之间的全连接实现，这样两个距离较远的点之间的连接就被相对弱化了，从而减少了运算量。卷积核的这种特点，可以被概括为权值共享，即卷积核作为媒介不断地在局部连接各个像素点，本身却只用一套参数，相当于是各个像素点在连接时共享了一套权值，而不是简单的全连接，如图 5-5 所示。

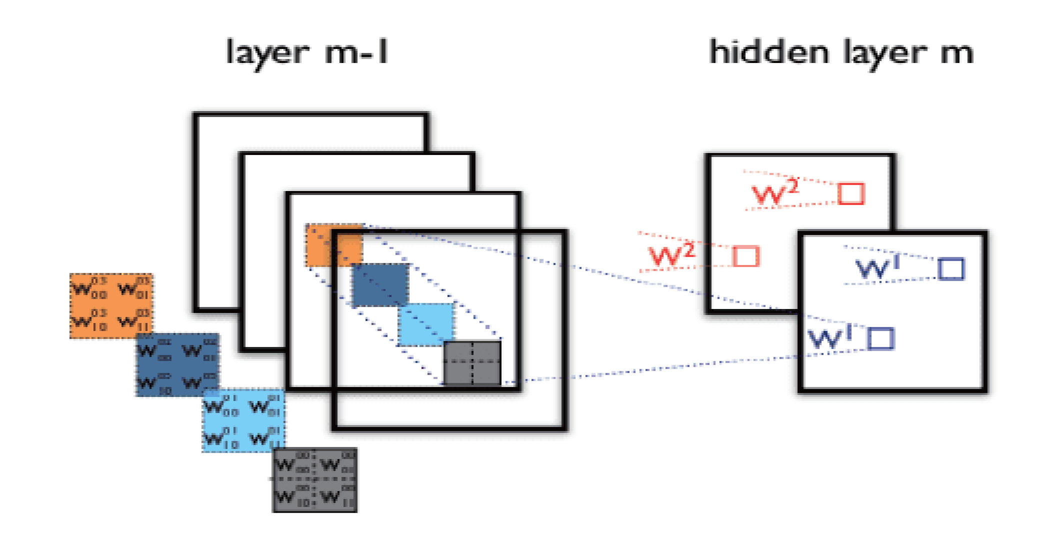


Figure 1: example of a convolutional layer

（图片来源：http://siliconmentor.blogspot.sg/2015/04/an-introduction-to-cnn.html）

图 5-5　卷积层对扫描结果进行了全连接组合

理解了卷积层并非只是扫描图像、还进行了图层整合这一点以后，看起来比较诡异的 1×1 卷积层是做什么的也就有了答案——这种情况就是只整合图层，不整合局部信息。通常情况下，如果 1×1 卷积层输入图层高于输出图层，这种情况就相当于对卷积层输出结果做了一次压缩。

讲完原理，再用代码加以巩固。我们将之前单颜色通道的输入图片，增加一个一模一样的颜色通道，这样矩阵的输入就成了 [1, 5, 5, 2]。由于图片输入通道增加，卷积核的输入通道也要进行相应的增加，除了之前的 3×3 矩阵之外，我们再引入一个卷积核：

0	1	0
1	0	1
0	1	0

扫描结果：

```
    # 考虑第二种卷积核的输出 y2_cov
    x_kernel_2 = np.array([
        [0,1,0],
        [1,0,1],
        [0,1,0]
    ], dtype=np.float32)

    tf_x_kernel_2 = tf.constant(np.reshape(x_kernel_2, newshape=[3,3,1,1]))
    y2 = tf.nn.conv2d(tf_x_input, tf_x_kernel_2, strides=[1,1,1,1],
padding="VALID")
    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        [y1_cov,y2_cov] = sess.run([y1, y2])

    print(u"第一种卷积核扫描结果：")
    print(y1_cov[0,:,:,0])
    print(u"第二种卷积核扫描结果：")
    print(y2_cov[0,:,:,0])
```

运算结果：

```
# out:
第一种卷积核扫描结果：
[[ 4.  3.  4.]
 [ 2.  4.  3.]
 [ 2.  3.  4.]]
第二种卷积核扫描结果：
[[ 2.  4.  2.]
 [ 2.  3.  4.]
 [ 2.  3.  2.]]
```

此时考虑为输入层添加一个图层，这样图层数从之前的一个增加到两个，图层的内容跟之前一样。接下来组合之前的两个卷积核，使组合后的卷积核可以接受两个图层输入，但只返回一层输出。发现其结果是两个图层分别扫描结果的简单相加：

```
    # 输入层加一个图层，里面内容同之前，这样输入维度增加到 2
    x_input2 = np.zeros([1,5,5,2])
    x_input2[0,:,:,0] = x_input
    x_input2[0,:,:,1] = x_input

    # 输入维度增加到 2，卷积核输入维度同样做相应的增加，但这里输出仍然是一个维度
    x_kernel_3 = np.zeros([3,3,2,1])
    x_kernel_3[:,:,0,0] = x_kernel_1
    x_kernel_3[:,:,1,0] = x_kernel_2

    tf_x_input2  = tf.constant(x_input2.astype(np.float32) )
    tf_x_kernel_3 = tf.constant(x_kernel_3.astype(np.float32) )

    y3 = tf.nn.conv2d(tf_x_input2, tf_x_kernel_3, strides=[1,1,1,1],
padding="VALID")
    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        [y3_cov] = sess.run([y3])
    # 发现这里卷积层输出内容是单独两个卷积核扫描结果的直接相加
    print(u"第一、第二种卷积核扫描结果简单相加：")
    print((y1_cov+y2_cov)[0,:,:,0])

    print(u"第一、第二种卷积核组合后扫描两层相同输入图层结果：")
    print(y3_cov[0,:,:,0])
```

运算结果：

```
# out:
第一、第二种卷积核扫描结果简单相加：
[[ 6.  7.  6.]
 [ 4.  7.  7.]
 [ 4.  6.  6.]]
第一、第二种卷积核组合后扫描两层相同输入图层结果：
[[ 6.  7.  6.]
 [ 4.  7.  7.]
 [ 4.  6.  6.]]
```

我们发现输入维度增加后，其实就是对两层的结果做了简单相加。我们进而将卷积层的输出也增加到两个维度，即卷积核的输出维度为 2，然后分别控制卷积核的输入使用相同卷积图层、卷积核的输出层使用相同的卷积图层：

```
    x_kernel_4 = np.zeros([3,3,2,2])
    x_kernel_4[:,:,0,0] = x_kernel_1
```

```
    x_kernel_4[:,:,1,0] = x_kernel_1
    x_kernel_4[:,:,0,1] = x_kernel_2
    x_kernel_4[:,:,1,1] = x_kernel_2

    x_kernel_5 = np.zeros([3,3,2,2])
    x_kernel_5[:,:,0,0] = x_kernel_1
    x_kernel_5[:,:,0,1] = x_kernel_1
    x_kernel_5[:,:,1,0] = x_kernel_2
    x_kernel_5[:,:,1,1] = x_kernel_2

    tf_x_kernel_4 = tf.constant(x_kernel_4.astype(np.float32))
    tf_x_kernel_5 = tf.constant(x_kernel_5.astype(np.float32))
    y4 = tf.nn.conv2d(tf_x_input2, tf_x_kernel_4, strides=[1,1,1,1],
padding="VALID")
    y5 = tf.nn.conv2d(tf_x_input2, tf_x_kernel_5, strides=[1,1,1,1],
padding="VALID")
    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        [y4_cov, y5_cov] = sess.run([y4, y5])

    print(u" 输出层用相同卷积核的结果 ")
    print(y4_cov[0,:,:,0])
    print(y4_cov[0,:,:,1])

    print(u" 输入层用相同卷积核的结果 ")
    print(y5_cov[0,:,:,0])
    print(y5_cov[0,:,:,1])
```

运算结果：

```
# out:
输出层用相同卷积核的结果：
[[ 8.  6.  8.]
 [ 4.  8.  6.]
 [ 4.  6.  8.]]
[[ 4.  8.  4.]
 [ 4.  6.  8.]
 [ 4.  6.  4.]]
输入层用相同卷积核的结果：
[[ 6.  7.  6.]
 [ 4.  7.  7.]
 [ 4.  6.  6.]]
[[ 6.  7.  6.]
 [ 4.  7.  7.]
 [ 4.  6.  6.]]
```

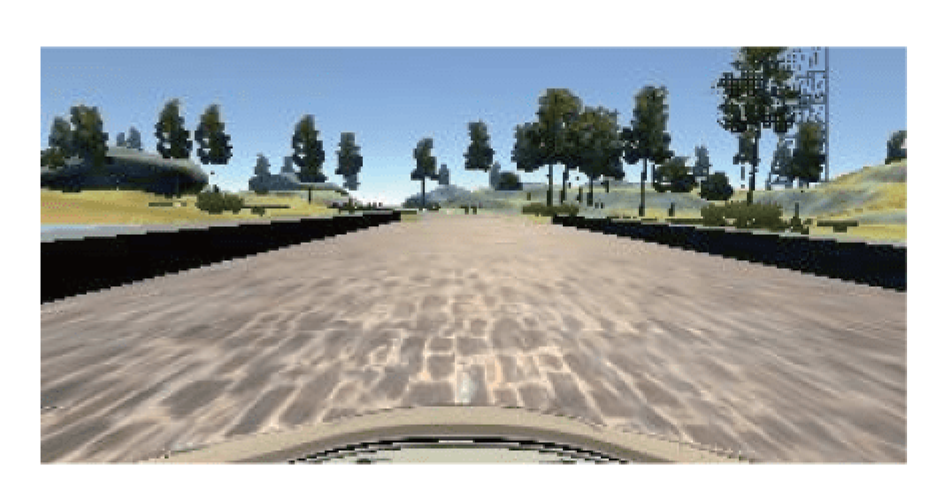
如果卷积核的输出层卷积通道相同、使用不同输入层的话，卷积层的两个输出将分别是两个卷积层扫描结果的两倍。如果卷积核的输入层卷积通道相同、使用不同输出层的话，则卷积层的两个输出就是两个输入通道扫描结果的直接相加之和。

3. 关于卷积核的第三种错误认识——随机初始完全随机

对于问题 3，这里涉及一些深度模型特有的问题。在逻辑回归中，初始化参数，我们可以猜一个，比如直接用 np.random.random(n)，但是如果模型很深、有连续的加法和乘法操作，此时随机引入的初始分布所带的方差会由于连乘操作逐级放大或者缩小，继而在计算反向梯度时干扰计算，因此需要一个合理的初始化方法，尽可能使随机引入的方差保持稳定。这里具体的初始化方法，大家可以阅读 Xavier 初始化的论文。

5.2.2 Cropping2D

这里 Cropping2D 就比较好理解了，就是特地选取输入图像的某一个固定的小部分。比如车载摄像头检测路面的马路线时，摄像头上半部分拍到的天空就可以被 Cropping2D 函数直接切掉忽略不计，如图 5-6 所示。



Original image taken from the simulator



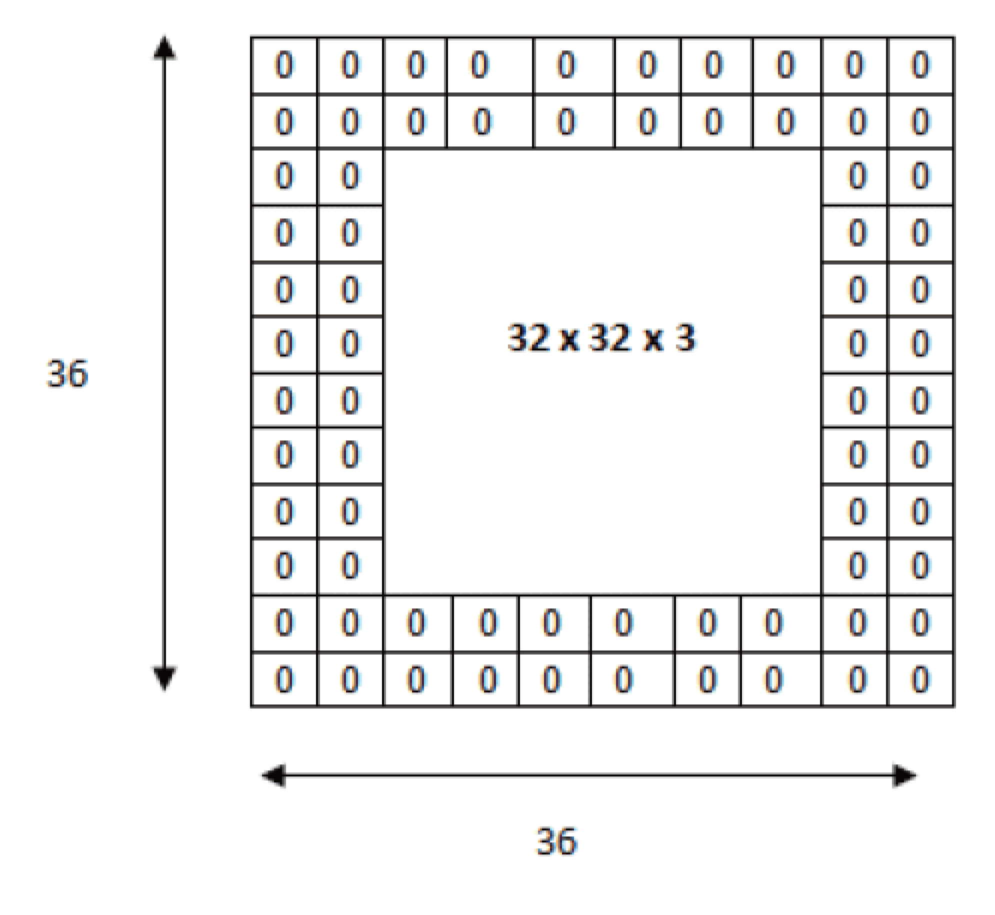
Cropped image after passing through a Cropping2D layer

（图片来源：Udacity 自动驾驶课程 https://www.udacity.com/drive）

图 5-6 使用 Cropping 层截取需要的区域进行进一步分析

5.2.3 ZeroPadding2D

2.2.1 节提到输入参数时，提到 padding 参数如果是 same，扫描图像边缘时会补上 0，确保输出数量等于 input / strides。这里 ZeroPadding2D 的作用就是在图像外层边缘补上几层 0。如图 5-7 所示，就是对原本 32×32×3 的图片进行 ZeroPadding2D(padding=(2, 2)) 操作后的结果。



（图片来源：Adit Deshpande 博客 https://adeshpande3.github.io/adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks-Part-2/）

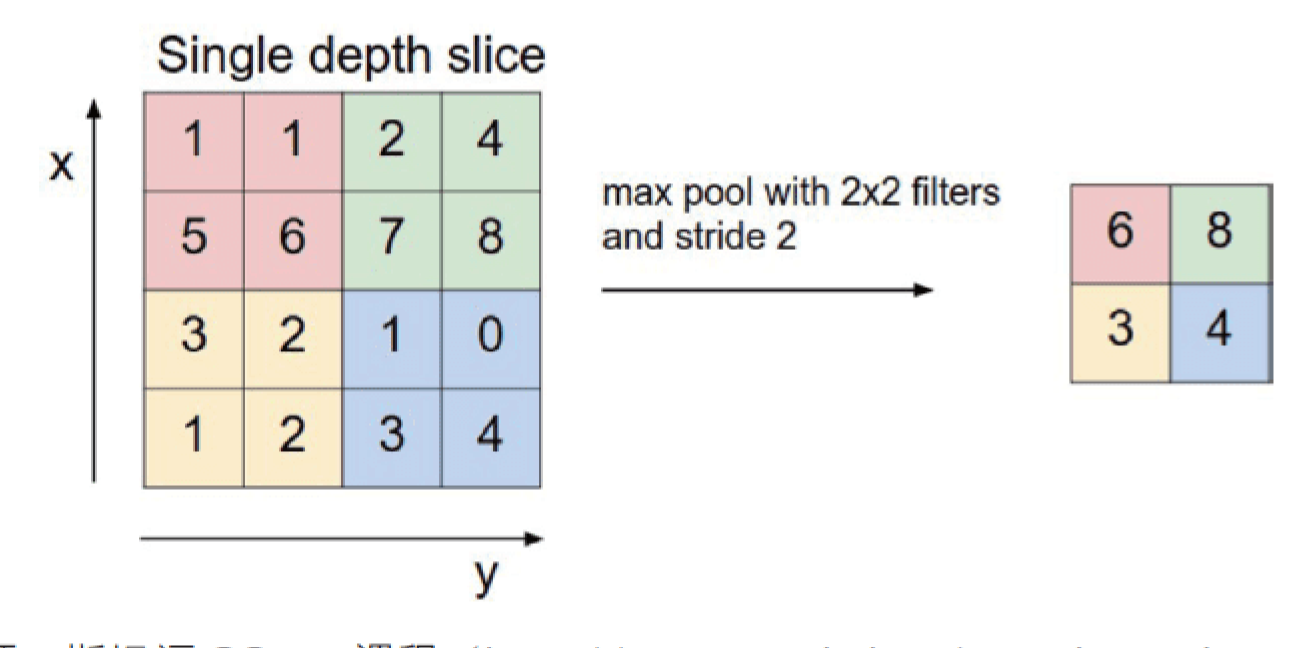
图 5-7 Padding 层

5.3 池化层

5.3.1 MaxPooling2D

通过与一个相同的、大小为 11×11 的卷积核做卷积操作，每次移动步长为 1，则相邻的结果会非常接近，正是由于结果接近，有很多信息是冗余的。

因此，最大池化（MaxPooling）就是一种减少模型冗余程度的方法。以 2×2 MaxPooling 为例。图中如果是一个 4×4 的输入矩阵，则这个 4×4 的矩阵会被分割成由两行、两列组成的 2×2 子矩阵，然后每个 2×2 子矩阵取一个最大值作为代表，由此得到一个两行、两列的结果，如图 5-8 所示。



（图片来源：斯坦福 CS231 课程（http://cs231n.github.io/convolutional-networks/））

图 5-8 最大池化层

5.3.2 AveragePooling2D

平均池化（AveragePooling）与最大池化类似，不同的是一个取最大值、一个取平均值。如果将图 5-8 中的 MaxPooling2D 换成 AveragePooling2D，结果如下：

3.25	5.25
2	2

在第 3 章讲解二维码压缩时，用到的就是这种方法。

5.3.3 GlobalAveragePooling2D

全局平均池化（GlobalAveragePooling，GAP）指的是之前举例平均池化提到的 2×2 池化，是对子矩阵分别平均，变成了对整输入矩阵求平均值。

这个理念其实和池化层关系并不十分紧密，因为它扔掉的信息有点过多了，通常只会出现在卷积神经网络的最后一层，是作为早期深度神经网络展开层（Flatten）+ 全连接层（Dense）结构的替代品，如图 5-9 所示。

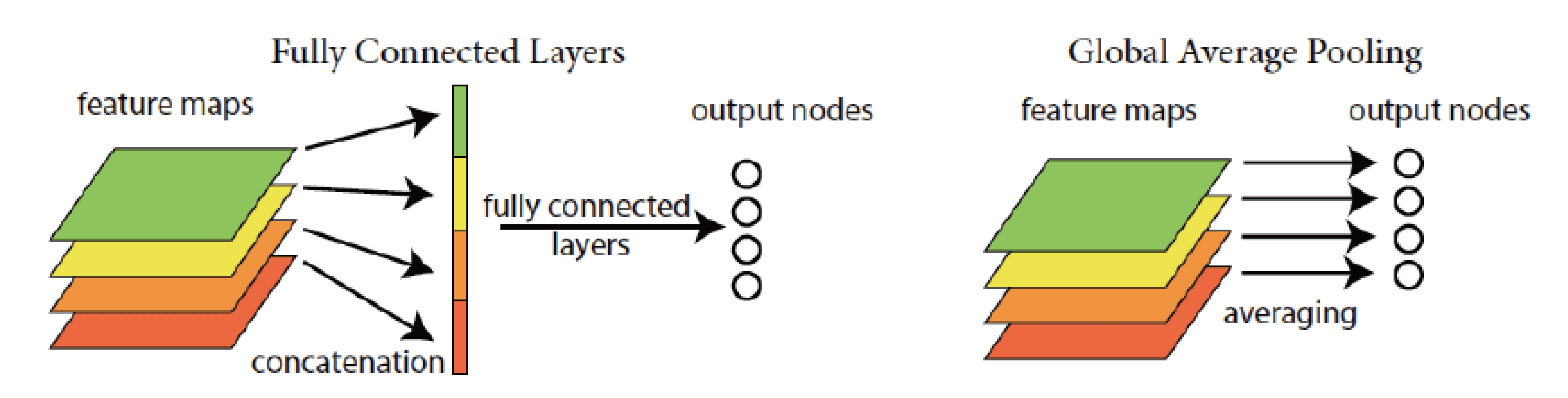


图 5-9　全局平均池化（图片来源：Network in Network）

前面提到过展开层通常位于连接深度神经网络的**卷积层部分**以及**全连接层部分**，但是这个连接有一个大问题，就是如果是一个 1k×1k 的全连接层，一下子就多出来百万参数，而这些参数实际用处相比卷积层并不高。造成的结果就是，早期的深度神经网络占据内存的大小反而要高于后期表现更好的神经网络，如图 5-10 所示。

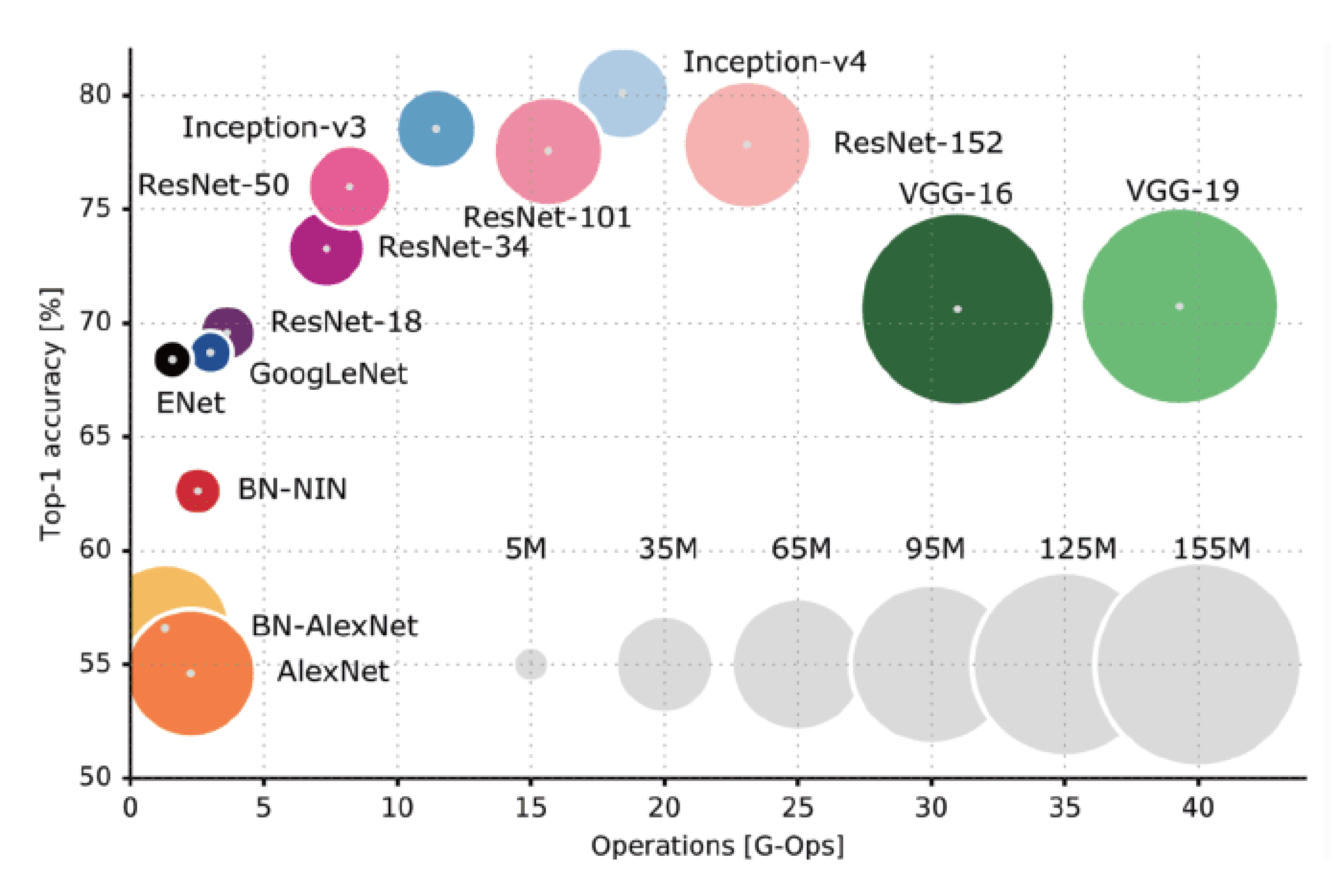


图 5-10　常见模型占用内存大小以及 Top1 分类准确率（图片来源：Training ENet on ImageNet）

更重要的是，全连接层由于参数偏多，更容易造成**过拟合**—— 前文提到的失活层就是为了避免过拟合的一种策略，进而由于过拟合妨碍整个网络的泛化能力。于是就有了用更多的卷积层提取特征，然后继续展开这些 k× k 大小卷积层，直接把这些 k×k 大小卷积层变成一个值，作为特征连接分类标签。

5.4 正则化层与过拟合

除了之前提到的失活策略、用全局平均池化层取代全连接层的策略，以及第 2 章提到的参数绝对值之和或平方和写进损失函数的 l1 l2 正则化之外，还有一种方法可以降低网络的过拟合，就是在深度神经网络中加入批正则化，这里着重介绍批正则化（Batch Normalization）。

当然，防止过拟合的方法还有很多，包括后面案例部分中，第 9、11 章用到的数据增强以及基于多任务学习的迁移学习，第 9 章用得多个深度神经网络结果模型平均，第 10 章训练 RNN 时，在调试阶段会用到的提前中止训练（见 https://ypwhs.github.io/captcha/），以及本书未涉及的更高级的策略，也包括对抗训练、稀疏编码、流形正切等。这些方法在这里不过多讨论，有兴趣继续学习的读者，可以仔细阅读 Deep learning 一书的第 7 章。

Batch Normalization

批正则化（Batch Normalization）提出的本意是为了加速神经网络训练的收敛速度。比如进行最优值搜索时，不清楚最优值位于哪里，可能是上千、上万，也可能是一个负数。这种不确定性会造成搜索时间的浪费，但实际应用在神经网络中时，发现这种方法实际上可以有效地降低过拟合。

批正则化层可以将需要进行最优值搜索数据转换成标准正态分布，这样 optimizer 就可以加速优化。算法如下：

输入：单个批次的输入数据：B

期望输出：β，γ

$$y_i = BN_{\gamma,\beta}(x_i)$$

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i$$

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2$$

$$\hat{\boldsymbol{x}}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

$$y = \gamma\hat{\boldsymbol{x}}_i + \beta$$

根据该算法，在批正则化层的训练过程中，将利用基于各个批次估计的数据的平均值、方差代替实际整体的平均值、方差，并根据估计出的平均值与方差将数据转换为标准正态分布，继而通过在训练过程中不断更新 γ 与 β 的值，将标准正态分布的数据进行还原，继而作为输出。

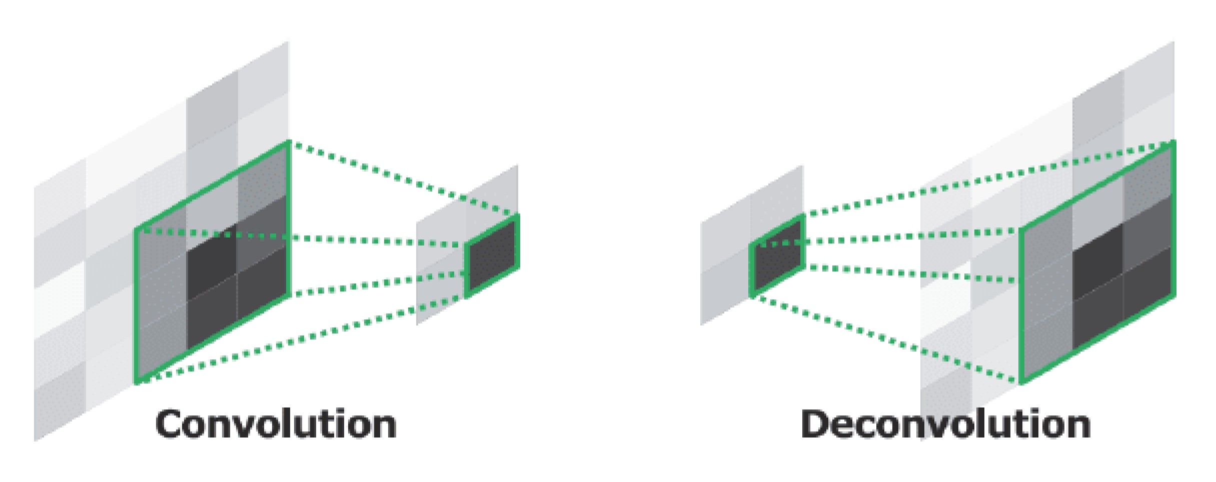
值得关注的是，dropout 与 batch_normalization 两种正则化手段，在 TensorFlow 的训练与预测过程中，均采用了不一样的策略。其中，dropout 在训练时将会随机失活一定百分比的神经元，而预测时为了避免随机选取带来的不确定性，将改为所有神经元数值减少该失活百分比；在 batch_normalization 中，训练阶段的数据平均值、方差是将当前批次值的平均值、方差以及估计的总体平均值、方差按照一定比例混合得到的，而预测阶段的平均值、方差则来自当前批次计算所得到的结果。

5.5 反卷积层

最后再谈一谈和图像分割相关的反卷积层。

之前介绍池化层时，在卷积神经网络中，池化可以通过对一片区域计算平均值、最大值，降低图片的大小，进而忽略无关信息。**换言之，卷积神经网络中的池化操作就是对输入图片打马赛克。**

马赛克是否有用？我们知道老程序员可以做到“图中有码，心中无码”，就是说，图片即便是打了马赛克、忽略了细节，仍然可以大概猜出图片的内容。这个过程就有点反卷积的意思了，如图 5-11 所示。



（图片来源：Learning Deconvolution Network for Semantic Segmentation）

图 5-11　卷积与反卷积过程

利用反卷积层，可以基于卷积层 + 全连接层结构构建新的**用于图像分割的神经网络**结构：

- 使用卷积 + 全连接层，如图 5-12 所示。

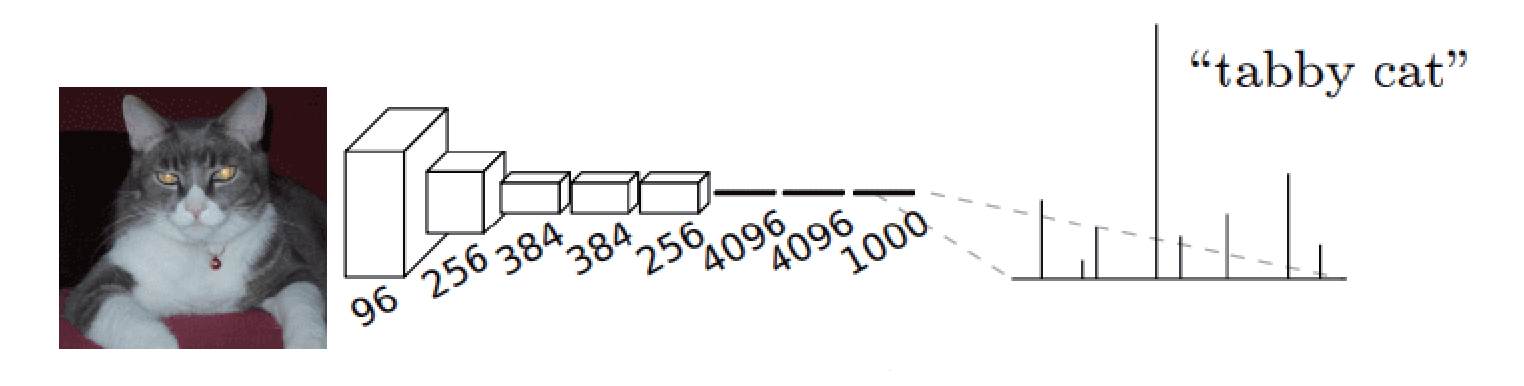
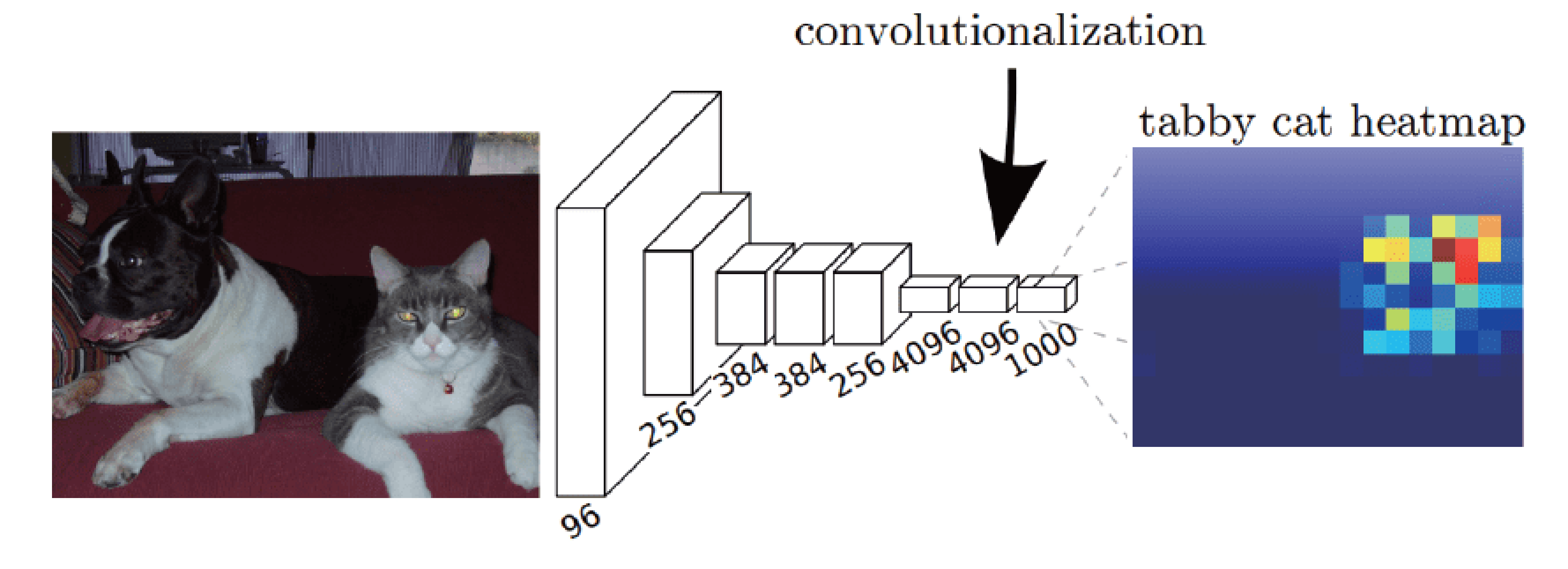


图 5-12　基于卷积 + 全连接层进行图像分类

- 全卷积层结构，使用卷积 + 反卷积层（Upsampling 过程），如图 5-13 所示。



（图片来源：Fully Convolutional Networks for Semantic Segmentation）

图 5-13　基于卷积 + 反卷积实现图像分类 + 定位

UpSampling2D

图 5-13 在最后阶段使用了上采样（Upsampling）模块，这个同样在 TensorFlow 的 Keras 模块可以找到。用法和 MaxPooling2D 基本相反，比如：

```
UpSampling2D(size=(2, 2))
```

相当于将输入图片的长宽各拉伸一倍，整个图片被放大了。

明白了大致原理，对这种复杂的模块，仍然是用底层 TensorFlow 代码巩固学习，用一个 3×3 的、全是 1 的卷积核，对输入层执行反卷积操作：

```
x_kernel_3 = np.array([
    [1,1,1],
    [1,1,1],
    [1,1,1]
], dtype=np.float32)
tf_x_kernel_3 = tf.constant(np.reshape(x_kernel_3, newshape=[3,3,1,1]))

y1_trans = tf.nn.conv2d_transpose(y1, tf_x_kernel_3, output_
shape=[1,5,5,1], strides=[1,2,2,1], padding="SAME")

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    [y1_cov_tran] = sess.run([y1_trans])

print(u" 反卷积输入 ")
print(y1_cov[0,:,:,0])
print(u" 反卷积输出 ")
print(y1_cov_tran[0,:,:,0])
```

运算结果：

```
# out:
反卷积输入：
[[ 4.  3.  4.]
 [ 2.  4.  3.]
 [ 2.  3.  4.]]
反卷积输出：
[[  4.   7.   3.   7.   4.]
 [  6.  13.   7.  14.   7.]
 [  2.   6.   4.   7.   3.]
 [  4.  11.   7.  14.   7.]
 [  2.   5.   3.   7.   4.]]
```

粗看起来，结果难以解释——5×5 矩阵，最中间的点以及最外层边缘、中间的 8 个点，与输入的卷积层一模一样，但其他点是怎么来的？其实这里反卷积首先进行了一个补 0 的操作，将输入变成：

4	0	3	0	4
0	0	0	0	0
2	0	4	0	3
0	0	0	0	0
2	0	3	0	4

再用 3×3、全是 1 的卷积核扫描，就得到了反卷积的结果：

```
    x_input_tran_zero = np.array([
            [4,0,3,0,4],
            [0,0,0,0,0],
            [2,0,4,0,3],
            [0,0,0,0,0],
            [2,0,3,0,4]
    ], dtype=np.float32)
    tf_input_tran_zero  = tf.constant(np.reshape(x_input_tran_zero,
newshape=[1,5,5,1]))
    y1_trans_zero = tf.nn.conv2d(tf_input_tran_zero, tf_x_kernel_3,
strides=[1,1,1,1], padding="SAME")

    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        [y1_cov_tran2] = sess.run([y1_trans_zero])

    print(u" 反卷积输出 ")
    print(y1_cov_tran2[0,:,:,0])
```

运算结果：

```
# out:
反卷积输出
[[  4.   7.   3.   7.   4.]
 [  6.  13.   7.  14.   7.]
 [  2.   6.   4.   7.   3.]
 [  4.  11.   7.  14.   7.]
 [  2.   5.   3.   7.   4.]]
```

这样就十分清楚了，反卷积的实质，其实就是将原矩阵扩大后中间补 0，然后对扩大后矩阵做卷积的过程。

通过反卷积将矩阵大小扩大到 5，并且将其继续扩大到 9：

```
    y1_trans = tf.nn.conv2d_transpose(y1, tf_x_kernel_3, output_
shape=[1,9,9,1], strides=[1,4,4,1], padding="SAME")

    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        [y1_cov_tran] = sess.run([y1_trans])

    print(u" 反卷积输入 ")
    print(y1_cov[0,:,:,0])
    print(u" 反卷积输出 ")
    print(y1_cov_tran[0,:,:,0])
```

运算结果：

```
# out:
反卷积输入
[[ 4.  3.  4.]
 [ 2.  4.  3.]
 [ 2.  3.  4.]]
反卷积输出
[[ 4.  4.  0.  3.  3.  3.  0.  4.  4.]
 [ 4.  4.  0.  3.  3.  3.  0.  4.  4.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 2.  2.  0.  4.  4.  4.  0.  3.  3.]
 [ 2.  2.  0.  4.  4.  4.  0.  3.  3.]
 [ 2.  2.  0.  4.  4.  4.  0.  3.  3.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 2.  2.  0.  3.  3.  3.  0.  4.  4.]
 [ 2.  2.  0.  3.  3.  3.  0.  4.  4.]]
```

由此可见，这个结果同样是原矩阵各元素等比例移动、其他位置补 0，得到一个大矩阵，然后对大矩阵做卷积得到的。

最后，反卷积层同样涉及图层之间输入输出维度的组合问题，这部分内容与卷积层相同，这里不再赘述。

5.6 循环层

循环层（Recurrent Neural Networks，RNN）常被用于处理有上下文或时间关系的内容，如语义理解、语音识别等，本书第 10 章的案例将使用循环层进行验证码识别。我们以人类阅读行为进行类比，在阅读文章时，虽然看的是当前文章中的这个词语，但实际上脑子里会记录之前看见的内容，具体可能是刚看过的两句话记得比较清楚，两分钟前看的只记得大致意思，再早看见的很多就忘了，而作者反复强调的内容或者比较重要的部分，可能过两天之后还能清楚记得。所以实际上人是基本做不到“过目不忘”的，但这并不影响人们进行阅读，学习新的内容，因为忘掉的部分并不重要，要点已经被记下来了。

循环神经网络就是用来处理数据中存在的上下文关系的。Keras 中常用的模块包括 SimpleRNN、LSTM 以及 GRU 三种。

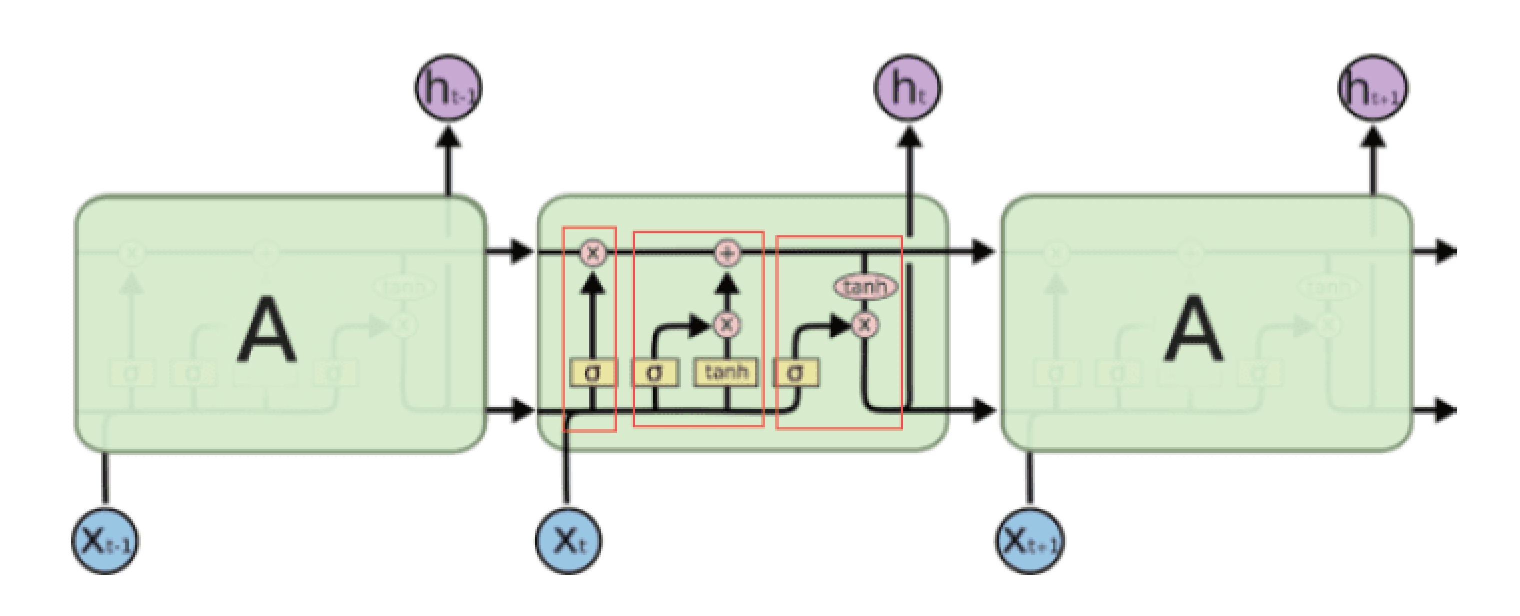
5.6.1 SimpleRNN

SimpleRNN 是 RNN 中的全连接网络。我们在 5.1 节说过全连接层就是进行简单的矩阵乘法，所以实际上 SimpleRNN 算法不存在“遗忘”功能，对历史数据必须“过目不忘”，每增加一个当前的状态，就需要再对前面一段时间点中各个历史状态建立连接。

这种方法的问题是，本书 5.2.1 节介绍卷积层的第三种错误认识时，提到合理的参数初始化很重要，因为深度神经网络中会有反复的连续乘法操作，此时一旦初始化做得不够好，随机引入的初始分布所带的方差会由于连乘操作逐级放大或者缩小，造成梯度爆炸或者梯度消失，特别是 RNN 处理历史数据的话，会根据不同时间点进行几十次、上百次连乘操作，远高于通常神经网络中十几次或者几十次的情况。在这种情况下，梯度爆炸、消失的问题会更加严重，因此通常并不使用 SimpleRNN，在使用循环神经网络时，通常用到的是下面介绍的 LSTM 和 GRU 两种。

5.6.2 LSTM

LSTM 为了解决 SimpleRNN 的问题，提出了一种记忆机制，通过三个门控单元来对历史信息单元的内容进行交互。LSTM 的组织结构如图 5-14 所示，每一个时间点的输入数据，分别对应一个 LSTM 单元。下间点对应 LSTM 单元的参数，即记忆单元。黄色框 tanh σ 分别是 tanh sigmoid 激活层（参见 5.1.2 节），红色圆框中 x 代表两个数做乘法运算、+ 代表两个数做加法运算。



（图片来源：Colah 博客（http://colah.github.io/posts/2015-08-Understanding-LSTMs/））

图 5-14 LSTM 所包含的各个单元

注意，由于 sigmoid 激活层在绝大多数情况下返回的是 0 或者 1，除非输入数字绝对值大于 5，因此可以认为**这几个激活函数控制的门非开即闭**：

- 输入门（从左到右第一个框）：输入门控制当前 xt 的信息融入记忆单元 ct，可以理解当前这个词的内容，对整句话是否重要。
- 遗忘门（从左到右第二个框）：遗忘门控制上一个时间点的记忆单元 ct-1 的内容融入当前记忆单元 ct，即当前内容是否同上文有关。如果相关，则梯度反向传播时就可以直接从当前层的 ct 传播至上一层的 ct-1，从而避免了很多连续的乘法运算，继而有效缓解了梯度消失。
- 输出门（从左到右第三个框）：输出门控制当前 ct 的信息融入隐层单元输出 ht，即这里判断了 ct-1 中的哪些内容对输出有用。

可以参考 https://zhuanlan.zhihu.com/p/28297161。

5.6.3 GRU

GRU 是 LSTM 的一种简化版本，具体而言有以下化简：

- 合并了 LSTM 的输入门和遗忘门，成为更新门。
- 合并了记忆单元 ct 和隐层单元 ht，仍然是隐层单元。

继而有：

- 重置门：用于控制前一时刻隐层单元 ht 对当前词 xt 的影响，即总体情况对当前状态的影响。
- 更新门：用于决定是否忽略当前词 xt，即当前状态是否影响总体情况。

可以参考 https://zhuanlan.zhihu.com/p/28297161。

5.7 参考文献及网页链接

［1］ 激活函数与 caffe 及参数 . fabulousli 的博客 . CSDN 博客 Available at: http://m.blog.csdn.net/ fabulousli/article/details/52890463.

［2］ An Introduction to CNN: Carrying the Machine learning on its shoulders. Available at: http://siliconmentor.blogspot.sg/2015/04/an-introduction-to-cnn.html

［3］ CS231n Convolutional Neural Networks for Visual Recognition. Available at: http://cs231n.github.io/convolutional-networks/.

［4］ Training ENet on ImageNet. Available at: https://culurciello.github.io/tech/2016/06/20 / training-enet.html.

［5］三次简化一张图：一招理解 LSTM/GRU 门控机制 . 知乎专栏 . Available at: https://zhuanlan.zhihu.com/p/28297161.

［6］ BigMoyan. Keras: 基于 Python 的深度学习库 . Keras 中文文档 . Available at: http://keras-cn.readthedocs.io/en/latest/.

［7］Deshpande, A. A Beginner’s Guide To Understanding Convolutional Neural Networks Part 2. A Beginner’s Guide To Understanding Convolutional Neural Networks Part 2 – Adit Deshpande – CS Undergrad at UCLA (‘19). Available at: https://adeshpande3.github.io/adeshpande3.github.io/A-Beginner’s-Guide-To-Understanding-Convolutional-Neural-Networks-Part-2/.

［8］ Goodfellow, I. J., Bengio, Y. & Courville, A. Deep learning. (The MIT Press, 2016).

［9］ Ioffe, S. & Szegedy, C. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. [1502.03167] Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift (2015). Available at: https://arxiv.org/abs/1502.03167.

［10］ KDnuggets. KDnuggets Analytics Big Data Data Mining and Data Science. Available at: http://www.kdnuggets.com/2016/11/intuitive-explanation-convolutional-neural-networks.html.

［11］ Krizhevsky, A., Sutskever, I. & Hinton, G. E. NIPS Proceedingsβ. ImageNet Classification with Deep Convolutional Neural Networks (2016). Available at: https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.

［12］ Lin, M., Chen, Q. & Yan, S. Network In Network. [1312.4400] Network In Network (2014). Available at: https://arxiv.org/abs/1312.4400 and http://docplayer.net/39678159-Network-in-network.html.

［13］ Self-Driving Car Engineer. Udacity Available at: https://www.udacity.com/drive.

［14］ Shelhamer, E., Long, J. & Darrell, T. Fully Convolutional Networks for Semantic Segmentation. [1605.06211] Fully Convolutional Networks for Semantic Segmentation (2016). Available at: https://arxiv.org/abs/1605.06211.

［15］ Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I. & Salakhutdinov, R. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. Journal of Machine Learning Research (2014). Available at: http://jmlr.org/papers/v15/srivastava14a.html.

第 6 章

少量多次——深度神经网络框架的输入处理

讲解完模型的基本模块后，再谈谈这些模块的上游和下游。我们可以利用这些模块来**组合更复杂的模型、挖掘更复杂的特征**。事实上，这几年深度学习领域的新进展就是以这个想法为基础产生的。我们可以使用更复杂的深度学习网络，在图片中挖出数以百万计的特征。

这时问题也就来了。机器学习过程中是需要一个输入文件的。这个输入文件的行、列分别指代样本名称以及特征名称。如果是进行百万张图片的分类，每个图片都有数以百万计的特征，我们将拿到一个**百万样本 × 百万特征**的巨型矩阵。传统的机器学习方法拿到这个矩阵时，受限于计算机内存大小的限制，通常是无从下手的。也就是说，传统的机器学习方法，除了在多数情况下不会自动产生这么多的特征以外，模型的训练也会是一个大问题。

深度学习算法为了实现对这个量级数据的计算，做了以下算法以及工程方面的创新：

- 将全部所有数据按照样本拆分成若干批次，每个批次大小通常在十几个到 100 多个样本之间（本章接下来要讲的内容）。
- 将产生的批次逐一参与训练，更新参数（下一章，深度神经网络框架的模型训练的内容）。
- 使用 GPU 等并行计算卡代替 CPU，加速并行计算速度。

这就有点“愚公移山”的意思了。我们可以把训练深度神经网络的训练任务，想象成是搬走一座大山。在成语故事中，愚公的办法是既然没有办法直接把山搬走，那就让子子孙孙每人每天搬几筐土走，山就会越来越矮，总有一天可以搬完——这种任务分解方式就如同深度学习算法的分批训练方式。同时，随着科技进步，可能搬着搬着就用翻斗车甚至是高科技来代替背筐，就相当于是用 GPU 等并行计算卡代替了 CPU。

我们接下来要讲的就是如何将深度学习框架运用在图像处理的使用场景中。实际工程环节，我们需要解决三个问题：

- 深度神经网络框架的图像输入接口怎么做？（怎么将大山分为小土堆）
- 深度神经网络框架的内部如何设计？
- 深度神经网络框架的参数如何优化？（怎么将土堆搬走）

这三个问题分别对应着深度神经网络的上游、深度神经网络本身以及深度神经网络的下游如何设计。本章首先来谈一谈上游部分，即如何设计深度神经网络框架的图像接口。

6.1 批量生成训练数据

在第 3 章中已经了解到，位图在计算机中通常以三维张量（Tensor）的形式存储，即 [图像高度，图像宽度，RGB 层]。而深度神经网络的接受数据输入，则是一个四维的张量，分别是 [图像编号，图像高度 , 图像宽度，RGB 层]，其中图像编号指的是，深度神经网络通常每次会接收多张图片，然后同时进行计算，这里的编号指代了某一图片是这一批图片中的第几张。

当我们的训练数据以一张张图片的形式保存在硬盘中时，如何实现一个函数，每调用一次，就会读取指定张数的图片（以 n=32 为例），将其转化成四维张量，返回输出，进而在接下来的环节中交给深度神经网络模型。

一个函数被调用才能执行，我们借助 Python 的生成器（generator）来实现。生成器的特点，这里我们借用廖雪峰博客中的一段话：

创建一个包含 100 万个元素的 list，不仅占用很大的存储空间，如果我们仅仅需要访问前面几个元素，那后面绝大多数元素占用的空间都白白浪费了。所以，如果 list 元素可以按照某种算法推算出来，那我们是否可以在循环的过程中不断推算出后续的元素呢？这样就不必创建完整的 list，从而节省大量的空间。在 Python 中，这种一边循环一边计算的机制称为生成器：generator。

概括而言，就是生成器可以节约内存占用——将成千上万甚至更多的图片保存在硬盘中就好，模型需要调用时再读入内存。我们首先来说一下如何将一个简单的函数改写为生成器。

原函数给定若干图片的存储位置，全部读入内存：

```
def get_image(path, shape=None):
    image = cv2.imread(path)
    image = image[:,:,::-1]
    if shape != None:
        image = cv2.resize(image, shape)
    return image

l_img = []
for imagepath in l_imagepath:
    img = get_image(imagepath)
    l_img.append(img)
```

改写成生成器，同样给定若干图片的存储位置，执行一次，只将 32 张图片读入内存：

```
from sklearn.utils import shuffle
def image_generator(pd_input, shape=(64,64), batch_size=32):
    num_samples = pd_input.shape[0]
    while 1:
        # 重排数据
        pd_input_shuffle = shuffle(pd_input)
        for offset in range(0, num_samples, batch_size):
            l_x = []
            l_y = []
            for idx in range(batch_size):
                batch_samples = pd_input_shuffle.iloc[offset:offset+ batch_size]
                try:
                    path  = batch_samples.iloc[idx]['Sample']
                    label = batch_samples.iloc[idx]['Class']
                    image = get_image(path, shape)
                    l_x.append(image)
                    l_y.append(label)
                except:
                    pass

            np_x = np.array(l_x)
            np_y = np.array(l_y)
            yield shuffle(np_x, np_y)

g = image_generator(pd_SampClass_train)
l_image,l_label = next(g)
```

实际应用中，Keras 也有生成器，可以直接使用 Keras 的生成器代码。注意实际使用的过程中，请在 data_path 下建立两个文件夹（二分类情形），分别将图片文件存入对应分类的文件夹：

```
train_generator = train_datagen.flow_from_directory(
        'data_path',
        target_size=(150, 150),
        batch_size=32,
        class_mode='binary')
```

Keras 有生成器，我们这里还要作为重点来说，是因为很多业务场景下，可以照搬别人高大上的神经网络架构、直接用 Keras 中各种复杂的优化器来训练网络，**反而是不怎么高大上的生成器需要数据科学家自己写**。在实际的图片分析中，很多时候手里的图像是需要进行预处理的，甚至这些输入图像并非常见的图像格式，例如 CT 医学影像数据。Keras 的函数虽然可以直接将图片以四维张量的形式输出，但如果输入的不是图片，而是 CT 数据这种复杂的格式，就需要自己写一个生成器。

最后，本章内容介绍的 Keras 生成器的写法，最终只会调用单个 CPU 进行图像预处理，这样可能会最终造成 CPU 图像预处理的速度低于 GPU 的模型训练速度。为了调用多个 CPU 进行图像预处理，一个简单的办法是，2.0.8 版本以上的 keras，在 model.fit_generator 函数中，可以指定 workers=6、use_multiprocessing=True，继而同时使用 6 个 CPU、调用 6 个生成器进行图像预处理（见本书 10.4.5 节以及对应代码 Lecture10/baiduyun_dl_competition/round2/test_fixedSize*.ipynb）。在此基础上，如果要进一步加速图像预处理速度，则需要对多个图像文件进行压缩处理、存储为一个二进制文件（如 TensorFlow 的 tf.record 文件）来加快读写速度，有兴趣的读者可以进一步地深入研究，这里不再详细介绍。

6.2 数据增强

数据增强（Data Augmentation）可以理解成对数据进行有放回的重新抽样。通过数据重抽样，可以在数据量较小的情况下更好地估计数据的分布情况。

如果在抽样的放回过程中对数据加入干扰因素，则这个过程就是数据增强。当然这里的干扰因素需要适度，这里适度的原则就是，对于人类专家而言，使用增强后的数据也可以做出正确的判断——比如一张汽车图片，我们局部放大一下，可能还认识这是汽车，那么这种增强的方法就可以；如果把这辆车的图片完全涂黑，自己都不认识了，那么这种增强方法就不可取。

常用的数据增强方法包括：

- 放大缩小图片
- 旋转图片
- 水平翻转图片

我们可以通过 Keras 的 ImageDataGenerator 来直接实现（见图 6-1）：

```
# 装载数据
l_imagepath = ['./dataset/vehicles/KITTI_extracted/4374.png']
l_img = []
for imagepath in l_imagepath:
    img = get_image(imagepath)
    l_img.append(img)

# 设置并初始化生成器
train_datagen = ImageDataGenerator(
    rescale=1.0/255,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True
)
train_generator = train_datagen.flow(np.array(l_img))
fig = plt.figure(figsize=(7,7))
ax = fig.add_subplot(3,3,1)
ax.imshow(l_img[0])
ax.set_axis_off()
ax.set_title("Raw Image")
for i in range(8):
    imgs = next(train_generator)
    ax = fig.add_subplot(3,3,i+2)
    ax.imshow(imgs[0])
    ax.set_axis_off()
    ax.set_title("Augmentation %d" % (i+1))
```

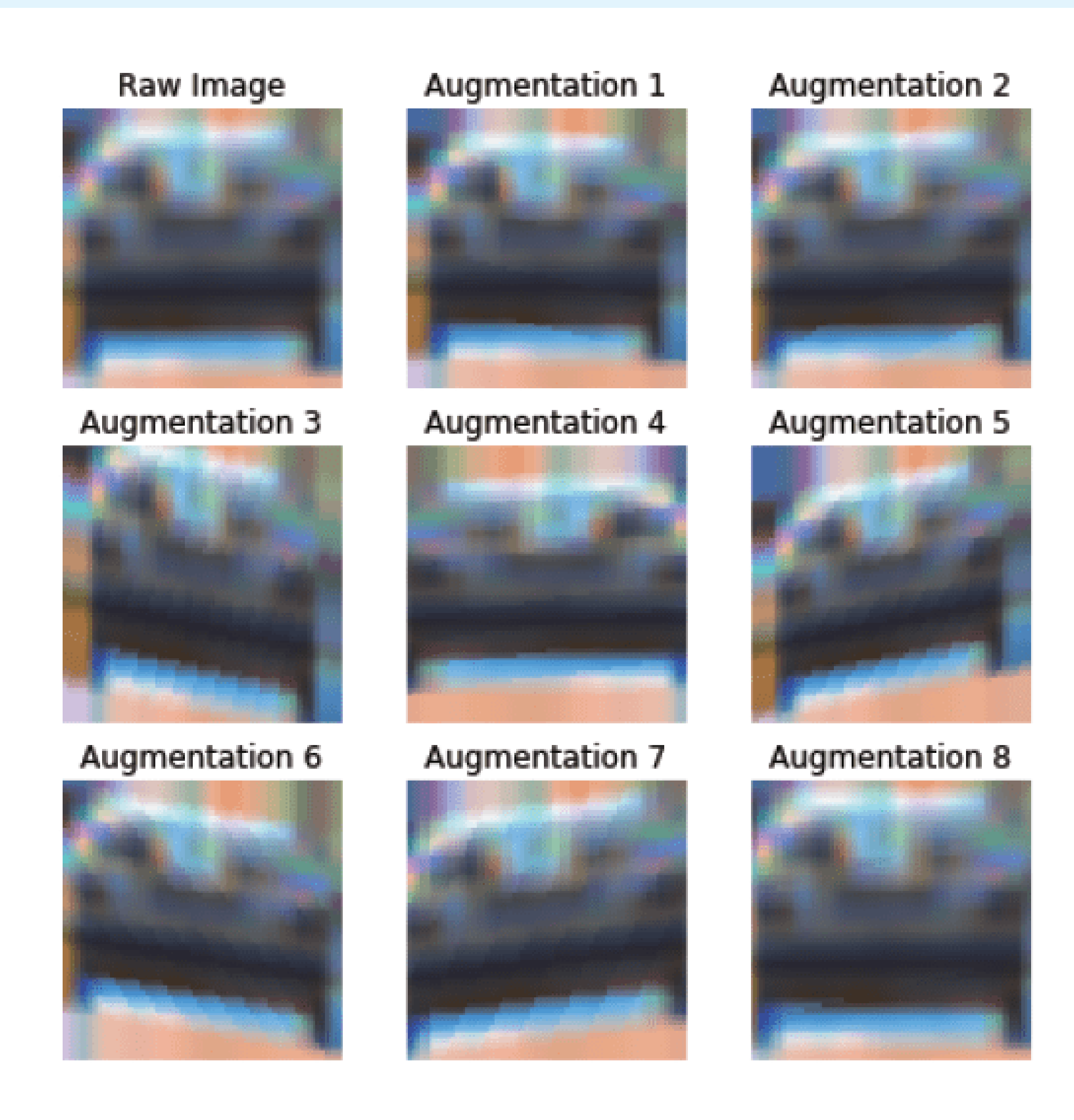


图 6-1　对单张图像（左上角）进行数据增强的结果

最后大家关注一下，我们使用了 rescale 参数，将输入图像中的数值从 int8 编码的 0~255 缩小到 0 和 1 之间的数字。当然，这里也可以换成标准正态分布。通过类似方法，可以有效地提升图像分类的准确性。具体原因类似在上一讲提到假如批正则化层，对数据进行正态分布转换、提升模型表现一样，深度神经网络由于层数很深，如果额外引入系统性的误差（如模型习惯处理标准正态分布输入，结果输入数据是 0~255 的图像），容易干扰模型稳定性，因此最好尽可能地让数值在一个较小的范围内降低模型在优化过程中的搜索空间。

6.3 参考文献及网页链接

［1］ Wong, S. C., Gatt, A., Stamatescu, V. & McDonnell, M. D. Understanding data augmentation for classification: when to warp? [1609.08764] Understanding data augmentation for classification: when to warp? (2016). Available at: https://arxiv.org/abs/1609.08764.

［2］生成器 . Home——廖雪峰的官方网站 . Available at: https://www.liaoxuefeng.com/wiki/0014316089557264a6b348958f449949df42a6d3a2e542c000/0014317799226173f45ce40636141b6abc8424e12b5fb27000.

第 7 章

愚公移山——深度神经网络框架的模型训练

本章将讲述如何使用基于批量梯度下降算法的凸优化模块，优化模型参数。

本章的标题虽然是“愚公移山”，但这并不代表这个步骤做的工作很机械、很简单。上一章提到，数据科学家在构建最初的模型过程中，主要时间花费在生成器的编写上，然后模型框架可以基于发表论文的经典模型微调，模型的参数优化可以直接用 TensorFlow、Keras 等框架写好的轮子。实际上，我们不花费大量时间编写这块内容的代码，并不代表不需要花费大量时间调试这块内容的参数。

需要花费大量精力调试这部分内容的原因很简单，很多情况下，我们写了成千上万行代码、搭建了一个模型的整体框架之后运行，模型不收敛，准确率长期得不到提升。此时可能改一下这部分内容涉及的几个参数之后，比如学习率以及第 5 章讲述卷积时提到常见错误 3 提到的参数的初始化方法，这个模型就能收敛。

在读者开始具体学习各种优化算法之前，为了防止读者被公式、代码整懵，先简单对优化模型这部分打个比方。常说深度学习如同“炼丹”，炼丹师借助了火的热量去炼丹，需要做的

只是控制“火候”，以及一次炼几个丹药而已，如同训练模型也不需要自己动手计算什么，全都交给计算机，但这并不代表炼丹师什么都没做，丹药就自己练出来了。控制火候、加药量是非常重要的经验。

我们炼丹的原则包括：

- 先用大火迅速进入状态，后用小火慢炖。
- 每次丹炼得越多，产量就越高，丹就炼得越快。但不要一次炼太多丹药，要和炼丹炉大小匹配，小炉子放过多丹药会影响产出品质。
- 每次炉子里要炼的丹加得多，火就要加得更旺，防止单个丹药分到的热量不足。

深度学习模型优化过程中，炼丹炉就是 GPU 等计算卡，火候就是学习率，每次炼几枚丹药就是批量大小。因此，接下来说的各种基于动量、自适应度调整学习率的算法，无非就是在实现炼丹原则 1——“先大火后小火”。

具体每次训练多少样本则取决于 GPU 显存一次能装下多少样本，如果是大量分布式训练，样本量就可以增加到几百、上千，但增加每次训练样本量（batch_size）的同时，学习率也需要有相应的上升。这部分内容算法没法帮助实现，主要需要读者根据自身机器性能进行选择。当然，这里并非机器越多、显存越大，只要调高并行度、增加学习率就可以无限地增加每次训练的样本量。目前，本章介绍的基于随机梯度下降的方法，对于 Resnet-50 模型，最多同时计算 8192 个样本，更大的批量将会导致训练无法收敛，结果准确率显著降低，因此已经有人开始尝试新的算法，如 Berkeley 提出的层级对应的适应率缩放算法（Layer-wise Adaptive Rate Scaling，LARS），就可以让批量大小扩大到更大的级别（如 32KB）而不损失结果准确度。读者有兴趣可以去阅读论文，我们这里提出这一点，只是为了让读者认识到虽然本章介绍的都是基于随机梯度下降的一系列方法，但是这些方法并非 全部。

最后，丹药有好炼的和难炼的种类，深度学习模型同样有容易训练的和较难训练的。容易训练的模型、数据，可能参数怎么设置，结果都会很好，区别不大，但这并不意味着所有模型都容易训练，结果都和参数无关，具体到实战时还有很多需要注意的地方。

7.1 随机梯度下降

前面提到，深度学习的“批量梯度下降”，可以理解为一群人拿着土筐，一点一点把山上的土给搬下山。那么这一点具体应该如何实现呢？其实在第 4 章就用 Python 实现了一个简单的随机批量梯度下降（Stochastic gradient descent, SGD），这里再回顾一下。这里随机梯度下降的算法是：

输入：参数 theta 学习率 learning_rate

while：学习过程：

  从训练集中取 m 个样本，作为小批量样本

  利用取出的样本，估计梯度 grad

  更新参数 theta = theta - learning_rate*grad

算法用代码实现如下：

```
def func_sgd(theta, eproch, xi, yi, zi, learning_rate):
    for i in range(eproch):
        grad = get_grad(xi, yi, zi, theta)
        theta = theta-grad*learning_rate

    return theta
```

7.2 动量法

单纯的随机梯度下降有一个问题，就是无法合理协调整体趋势和局部梯度之间的关系。《深度学习轻松学》一书中做了一个非常形象的比喻，就是极限运动的 U 形赛道，如图 7-1 所示。



（图片来源：http://img.bendibao.com/shanghai/201110/24/20111024165641447.JPG）

图 7-1 随机梯度下降寻找最小值，在优化 U 型地貌时收敛效率不高

这里的总体趋势是从后往前，但实际上左右之间的梯度是远高于前后之间的，所以随机梯度下降过程中，模型会过多地被左右方向的局部梯度带着走。

为了在优化过程中也考虑到总体的进程，我们这里引入动量来代表这种整体性：

输入：参数 theta，学习率 learning_rate，动量参数 alpha，初始速度 v

while：学习过程：

　　从训练集中取 m 个样本，作为小批量样本

　　利用取出的样本，估计梯度 grad

　　更新速度 v = alpha*v - learning_rate*grad

　　更新参数 theta = theta + v

算法用代码实现如下：

```
    def func_momentum(theta, eproch, xi, yi, zi, learning_rate, alpha,
velocity):
        for i in range(eproch):
            grad = get_grad(xi, yi, zi, theta)
            velocity = alpha*velocity - grad*learning_rate
            theta   = theta + velocity

        return theta
```

7.3 自适应学习率算法

我们注意到动量法中的参数更新环节不再是直接减去梯度，而是在梯度中引入了一个动量。这样在参数更新时，减去梯度的同时，也会加上动量的大小。更重要的是，这里动量在更新时会乘以一个衰减系数 alpha，这样实际上越到后来，参数的更新幅度就越小，保证了“先用大火后用小火”的原则。

实际上为了实现这个原则，我们既可以在学习率中引入动量，也可以直接控制学习率的大小，但动量法为了让 learning_rate 这个参数更加容易调试，又额外引进了其他的超参数 alpha，还给调参人员增加了一个需要测试的参数。

因此为了简化调参，就有算法试图找出可以直接迭代得到最合理的学习率的算法，这一系列算法被称作自适应学习率算法。自适应学习率算法背后的思路，背后有工科 PID 控制器的思维方式。

这种思维方式是，比如我们的炼丹炉，目标是维持温度在 600°，然后有一个加热器和一个传感器。于是希望有一个控制器，在炼丹炉温度低的情况下让加热器加热，在炼丹炉温度高的情况下让加热器停止加热，炼丹炉接近目标温度时会调低加热器的加热功率大小，但如果希望这个调整过程中温度变化平缓、减少波动，可能就不止需要考虑当前温度和目标温度的差距（比

例 Proportional，P），还需要考虑这一差距在过去时间内的累计积分（积分 Integral，I）、不同时间点之间的变化率（微分 Derivative，D）。于是可以引入一个 PID 控制器，这里 PID 控制器的含义及其使用方法，用 Python 描述如下：

```
class PID(object):
    # pid_p：当前误差。如目标是 600 度，现在 500 度，p 就是 100
    # pid_i: 累计误差。p 在时间上的累计之和。
    # pid_d: 误差变化。当前 p 同之前时间点 p 做减法。
    def __init__(self, kp, ki, kd):
        self.kp = kp
        self.ki = ki
        self.kd = kd
        self.pid_p = self.pid_i = self.pid_d = 0

    def update_error (self, error, sample_time):
        self.pid_d = error - self.pid_p
        self.pid_p = error
        self.pid_i += sample_time

def get_update(self):
   return self.pid_p * self.Kp + \
          self.pid_i * self.Ki + self.pid_d * self.Kd

# 初始化 PID 控制器 , kp, ki, kd 三个系数需要根据实际情况调整，达到最优的控制效果
Pid_heater = PID(kp=kp_init, ki=ki_init, kd=kd_init)

# 对每个时间点， 更新 PID 值，包括两步操作：
# 1. delta_T 是目标炉温和当前炉温的差距 , 认为每秒更新一次
Pid_heater.update_error(delta_T, sample_time=1)
# 2. PID 控制器返回加热器功率大小
double heater_value = Pid_heater.get_error();
```

如果将梯度理解成当前参数同最优参数之间的误差，那么其实可以用 PID 控制器的思想来改变学习率。这里我们介绍 AdaGrad 以及 Adam 两种算法。

（1）AdaGrad 算法考虑的是 PID 中的 PI 两项，即当前误差、累计误差。累计误差越大，学习率越小。

输入：参数 theta，学习率 learning_rate，小常数 delta

设置累计误差量 r = 0

while：学习过程：

从训练集中取 m 个样本，作为小批量样本

利用取出的样本，估计梯度 grad

更新梯度的累计平方误差， r = r + grad x grad（向量叉乘，各元素乘方后相加）

更新参数 theta -= learning_rate x grad / (delta+sqrt(r))

算法用代码实现如下：

```
def func_adagrad(theta, eproch, xi, yi, zi, learning_rate, delta):
    r = 0
    for i in range(eproch):
        grad = get_grad(xi, yi, zi, theta)
        r += np.dot(grad, grad)
        theta = theta - learning_rate/(delta+np.sqrt(r)) * grad

    return theta
```

（2）Adam 算法，同样考虑的是 PID 中的 PI 两项，不过这里 I 既跟 AdaGrad 算法一样计算了平方误差，同样也考虑了累计的一阶误差，更新方法也稍微复杂一些：

输入：参数 theta，学习率 learning_rate，小常数 delta

指数衰减速率 rho1 rho2，

迭代次数 t = 0

while：学习过程：

从训练集中取 m 个样本，作为小批量样本

利用取出的样本，估计梯度 grad

t = t+1

更新有偏一阶矩估计，s = rho1*s + (1-rho1)*grad

更新有偏一阶矩估计，r = rho2*r + (1-rho2)*grad x grad

修正一阶矩偏差：s = s / (1-rho1^t)

修正二阶矩偏差：r = r / (1-rho2^t)

更新参数 theta -= learning_rate * s / (delta+sqrt(r))

算法用代码实现如下：

```
def func_adam(theta, eproch, xi, yi, zi, learning_rate, delta, rho1,
rho2):
    s = 0
    r = 0
```

```
        for i in range(eproch):
            grad = get_grad(xi, yi, zi, theta)
            t = i+1
            s = rho1 * s + (1-rho1) * grad
            r = rho2 * 2 + (1-rho2) * np.dot(grad, grad)
            s_hat = s / (1-rho1**t)
            r_hat = r / (1-rho2**t)

            theta  = theta- learning_rate/(delta+np.sqrt(r_hat)) * s_hat

        return theta
```

Adam 优化器相对好控制，通常一个模型完成后可以设置 learning_rate=1e-3 的学习率，先跑一个基准模型，看看 3~5 个 batch 以内是否有收敛迹象。如果不收敛，就降低学习率到 1e-4、1e-5 继续训练。假如还不收敛，就去检查模型是否使用了合理的初始化（如 xavier_init），数据输入时是否进行标准化（图像值除以 255，或者变成标准正态分布），再有问题，最后考虑换个网络架构，以及数据量是否偏少。一旦模型收敛、loss 值下降就可以进一步通过微调参数、增加正则化等方法进行模型优化，继而考虑训练多个模型进行融合等。

7.4 实验案例

这里用之前定义的各个优化器来做一个“爬山游戏”。首先通过 scipy.interpolate.rbf 随机生成一个等高线图，如图 7-2 所示。

```
    import numpy as np
    from scipy.interpolate import Rbf
    import matplotlib.pyplot as plt
    %matplotlib inline

    np.random.seed(1981)
    x, y, z = np.random.random((3, 10))
    xi, yi = np.mgrid[0:1:100j, 0:1:100j]
    func = Rbf(x, y, z, function='linear')
    zi = -1*func(xi, yi)

    fig, ax = plt.subplots()
    dy, dx = np.gradient(-zi.T)
    contours = ax.contour(xi, yi, zi[:,::-1], linewidths=2)
    ax.clabel(contours)
    plt.show()
```

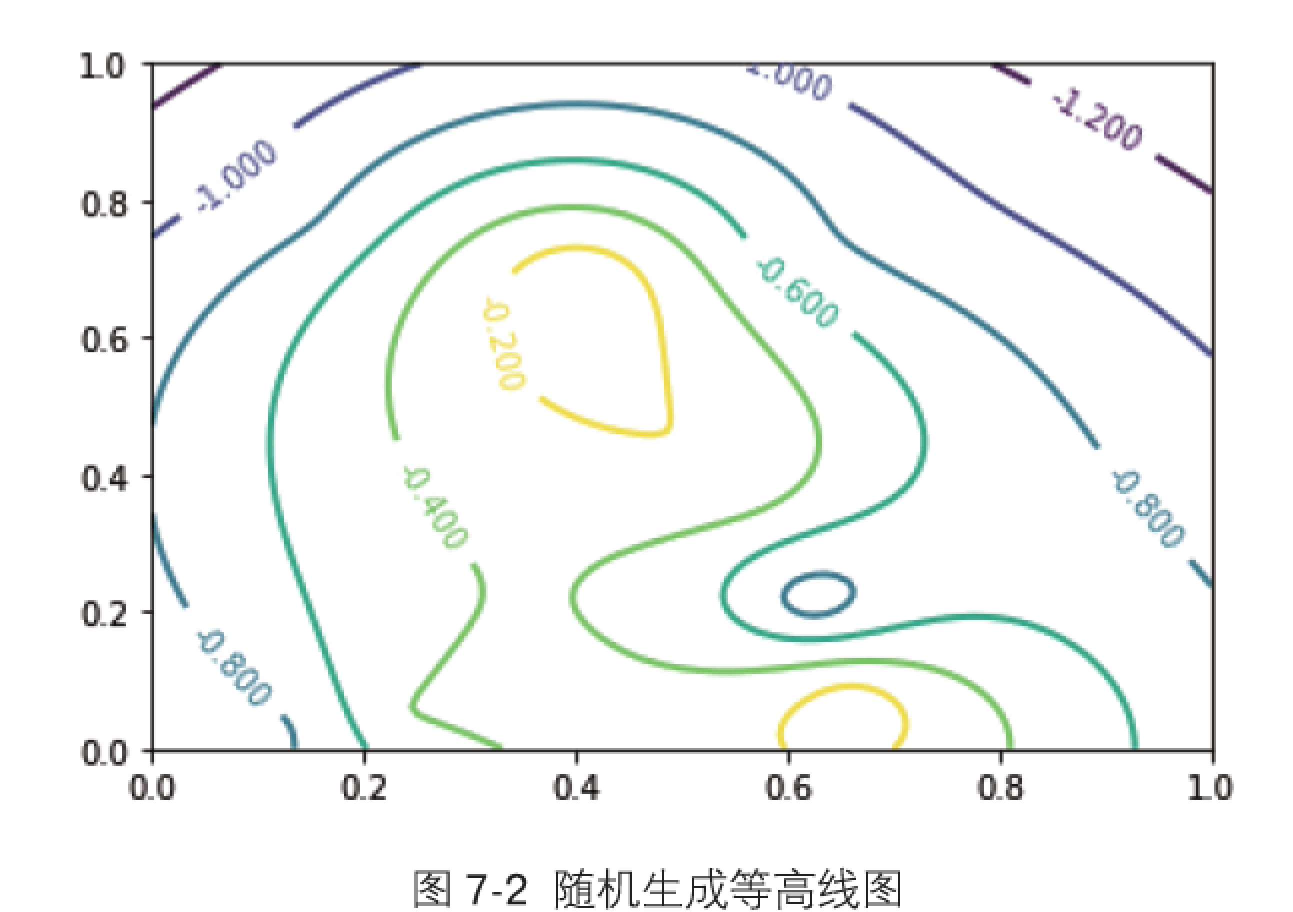


图 7-2　随机生成等高线图

我们的目的就是从任意一点起始，借助前面定义的优化器，**上中间黄色的高地**。借助等高线图，可以知道当前位置的高度（zi），以及周围 x、y 方向梯度的大小（dx, dy）。将这两个信息交给前面定义的优化器，看看结果如何（见图 7-3）。注意这里几个优化器函数相比之前的定义都有所改进，返回的不是最终所在的点，而是优化过程中的完整路径。

```
    position_init = np.array([0.7, 0.6])
    velocity = np.array([-0.01, -0.01])
    alpha = 0.5
    learning_rate = 0.05
    eproch = 10000
    delta = 1e-7
    rho1 = 0.9
    rho2 = 0.999

    def func_sgd(theta, eproch, xi, yi, zi, learning_rate):
        l_posx = []
        l_posy = []
        for i in range(eproch):
            grad = get_grad(xi, yi, zi, theta)
            theta = theta-grad*learning_rate
            l_posx.append(theta[0])
            l_posy.append(1-theta[1])

        return l_posx, l_posy

    def func_momentum(theta, eproch, xi, yi, zi, learning_rate, alpha,
velocity):
        l_posx = []
        l_posy = []
        for i in range(eproch):
            grad = get_grad(xi, yi, zi, theta)
```

```
            velocity = alpha*velocity - grad*learning_rate
            theta  = theta + velocity
            l_posx.append(theta[0])
            l_posy.append(1-theta[1])

        return l_posx, l_posy

    def func_adagrad(theta, eproch, xi, yi, zi, learning_rate, delta):
        l_posx = []
        l_posy = []
        r = 0
        for i in range(eproch):
            grad = get_grad(xi, yi, zi, theta)
            r += np.dot(grad, grad)
            theta  = theta- learning_rate/(delta+np.sqrt(r)) * grad
            l_posx.append(theta[0])
            l_posy.append(1-theta[1])

        return l_posx, l_posy

    def func_adam(theta, eproch, xi, yi, zi, learning_rate, delta, rho1,
rho2):
        l_posx = []
        l_posy = []
        s = 0
        r = 0
        for i in range(eproch):
            grad = get_grad(xi, yi, zi, theta)
            t = i+1
            s = rho1 * s + (1-rho1) * grad
            r = rho2 * 2 + (1-rho2) * np.dot(grad, grad)
            s_hat = s / (1-rho1**t)
            r_hat = r / (1-rho2**t)

            theta  = theta- learning_rate/(delta+np.sqrt(r_hat)) * s_hat
            l_posx.append(theta[0])
            l_posy.append(1-theta[1])

        return l_posx, l_posy

    l_posx_gd,l_posy_gd = func_sgd(
                              position_init, eproch,
                              xi, yi, zi, learning_rate)
```

```
    l_posx_momentum,l_posy_momentum = func_momentum(
                            position_init, eproch,
                            xi, yi, zi, learning_rate, alpha, velocity)

    l_posx_adagrad,l_posy_adagrad = func_adagrad(
                            position_init, eproch,
                            xi, yi, zi, learning_rate, delta)

    l_posx_adam,l_posy_adam = func_adam(
                            position_init, eproch,
                            xi, yi, zi, learning_rate, delta, rho1, rho2)

    fig, ax = plt.subplots()
    contours = ax.contour(xi, yi, zi[:,::-1], linewidths=2)
    ax.clabel(contours)
    ax.plot(l_posx_gd, l_posy_gd, ".", label="SGD")
    ax.plot(l_posx_momentum, l_posy_momentum, "g.", label="Momentum")
    ax.plot(l_posx_adagrad, l_posy_adagrad, "y.", label="Adagrad")
    ax.plot(l_posx_adam,     l_posy_adam,     "c.", label="Adam")

    ax.plot(l_posx_gd[-1],        l_posy_gd[-1],        "ro")
    ax.plot(l_posx_momentum[-1], l_posy_momentum[-1], "ro")
    ax.plot(l_posx_adagrad[-1],  l_posy_adagrad[-1],  "ro")
    ax.plot(l_posx_adam[-1],     l_posy_adam[-1],     "ro")
    ax.legend()
    plt.show()
```

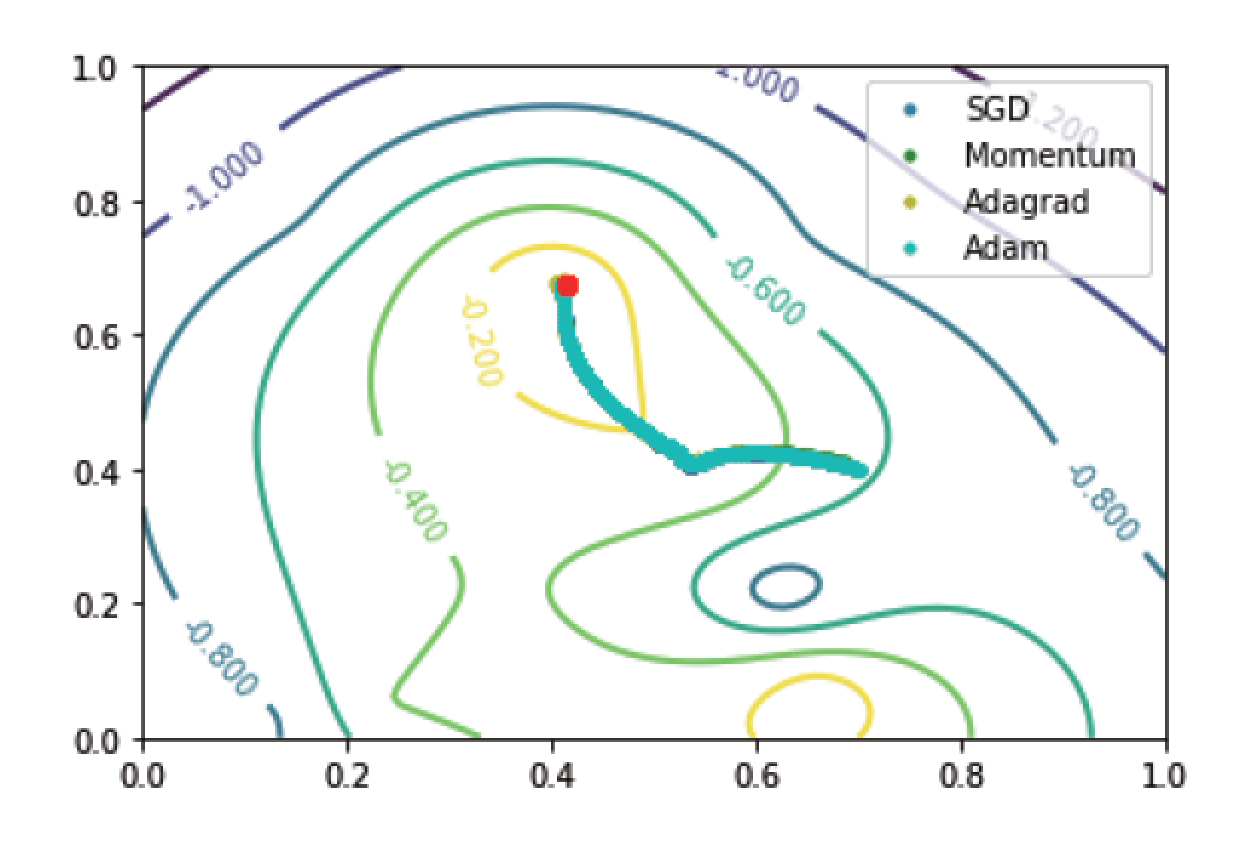


图 7-3　使用不同的优化器寻找等高线图最高点

可见从 [0.7, 0.6] 处起始，使用 0.05 的学习率，经过 10 000 次迭代后，几种算法都成功上了最高点，并且路径也十分接近。这里留给读者两个思考问题：

- 这些优化器的收敛速度有什么区别（试试更少的迭代次数，几种算法跑了多少）？
- 改变起始位置出发，结果有什么区别？

学有余力的读者也可以上网查一下如何用 matplotlib 制作动态图，动态绘制各优化器生成的路径，得到更好的可视化结果。

7.5 参考文献及网页链接

［1］ Goodfellow, I. J., Bengio, Y. Courville, A. Deep learning. (The MIT Press, 2016).

［2］ Stochastic gradient descent. Wikipedia (2017). Available at: https://en.wikipedia.org/wiki/Stochastic_gradient_descent#cite_note-2.

［3］ You, Y., Zhang, Z., Hsieh, C.-J. & Demmel, J. 100-epoch ImageNet Training with AlexNet in 24 Minutes. [1709.05011] 100-epoch ImageNet Training with AlexNet in 24 Minutes (2017). Available at: https://arxiv.org/abs/1709.05011.

［4］ 冯超 . 深度学习轻松学 [M]. 北京：电子工业出版社，2017.

第 8 章

小试牛刀——使用深度神经网络进行 CIFAR-10 数据分类

本章用一个 Keras 中的官方示例（https://github.com/fchollet/keras/blob/master/examples/ cifar10_cnn.py）来结束基础部分的内容。我们的分类对象是 CIFAR-10 数据集。这个数据集包含了 6 万张大小为 32×32 的彩色图片，其中 50000 张作为训练集、10000 张作为测试集，这些图片进而可以分成 10 个种类，如图 8-1 所示。

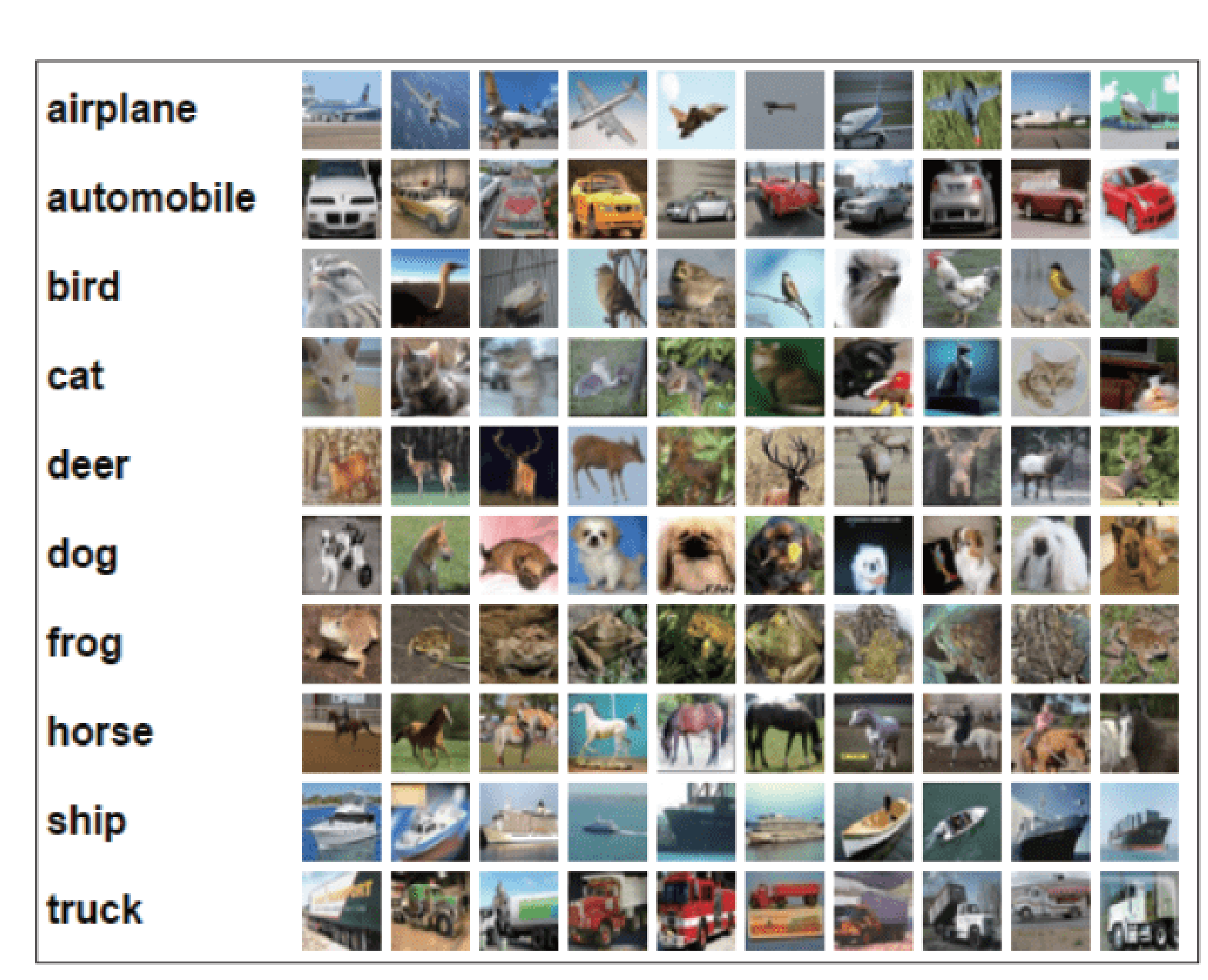


图 8-1 CIFAR-10 数据集的 10 个分类种类及其对应的实例图片

本章的案例就是使用 Keras 构建深度神经网络模型，然后基于构建好的深度神经网络模型对这个数据集进行训练，最后检验模型预测的准确性。

首先进行一些前期准备，调用 numpy 以及 Keras 中的相关函数：

```
# 初始化
from __future__ import print_function
import numpy as np
from keras.callbacks import TensorBoard
from keras.models import Sequential
from keras.optimizers import Adam
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Conv2D, MaxPool2D
from keras.utils import np_utils
from keras import backend as K
from keras.callbacks import ModelCheckpoint
from keras.preprocessing.image import ImageDataGenerator
from keras.datasets import cifar10

from keras.backend.tensorflow_backend import set_session
import tensorflow as tf
config = tf.ConfigProto()
config.gpu_options.allow_growth=True
set_session(tf.Session(config=config))
np.random.seed(0)
# 定义变量
batch_size = 32
nb_classes = 10
nb_epoch = 50
img_rows, img_cols = 32, 32
nb_filters = [32, 32, 64, 64]
pool_size = (2, 2)
kernel_size = (3, 3)

(X_train, y_train), (X_test, y_test) = cifar10.load_data()
X_train = X_train.astype("float32") / 255
X_test  = X_test.astype("float32") / 255

y_train = y_train
y_test = y_test

input_shape = (img_rows, img_cols, 3)
Y_train = np_utils.to_categorical(y_train, nb_classes)
Y_test = np_utils.to_categorical(y_test, nb_classes)
```

接下来的部分将对照本书第 5 章开头部分提到的三段论式架构，继而对照着第 6 章、第 5 章、第 7 章的内容介绍，分别构建：

- 处理的内容**是什么**——构建基于生成器的批量生成输入模块。
- **为什么**可以得到这样的结果——用各种零件搭建深度神经网络。
- 训练过程应该**怎么做**——使用凸优化模块训练模型。

8.1 上游部分——基于生成器的批量生成输入模块

本节将基于生成器的批量生成输入模块：

```
datagen = ImageDataGenerator(
            featurewise_center=False,
            samplewise_center=False,
            featurewise_std_normalization=False,
            samplewise_std_normalization=False,
            zca_whitening=False,
            rotation_range=0,
            width_shift_range=0.1,
            height_shift_range=0.1,
            horizontal_flip=True,
            vertical_flip=False)

datagen.fit(X_train)
```

8.2 核心部分——用各种零件搭建深度神经网络

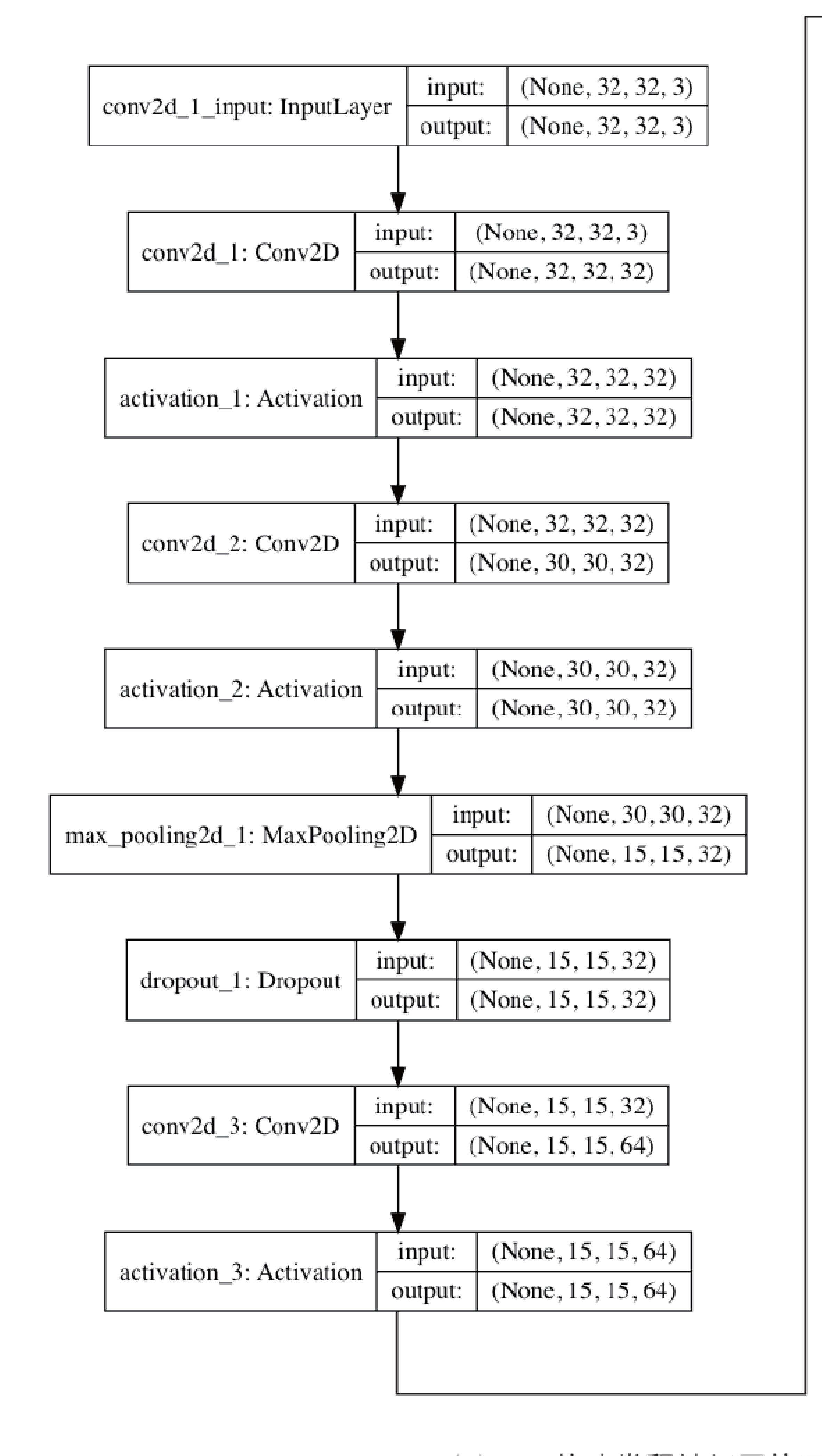
本节将用各种零件搭建深度神经网络：

```
model = Sequential()
model.add(Conv2D(nb_filters[0], kernel_size, padding='same', input_
shape=X_train.shape[1:]))
model.add(Activation('relu'))
model.add(Conv2D(nb_filters[1], kernel_size))
model.add(Activation('relu'))
model.add(MaxPool2D(pool_size=pool_size))
model.add(Dropout(0.25))

model.add(Conv2D(nb_filters[2], kernel_size, padding='same'))
model.add(Activation('relu'))
model.add(Conv2D(nb_filters[3], kernel_size))
model.add(Activation('relu'))
model.add(MaxPool2D(pool_size=pool_size))
model.add(Dropout(0.25))
```

```
model.add(Flatten())
model.add(Dense(512))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(nb_classes))
model.add(Activation('softmax'))
```

构建的模型如图 8-2 所示。



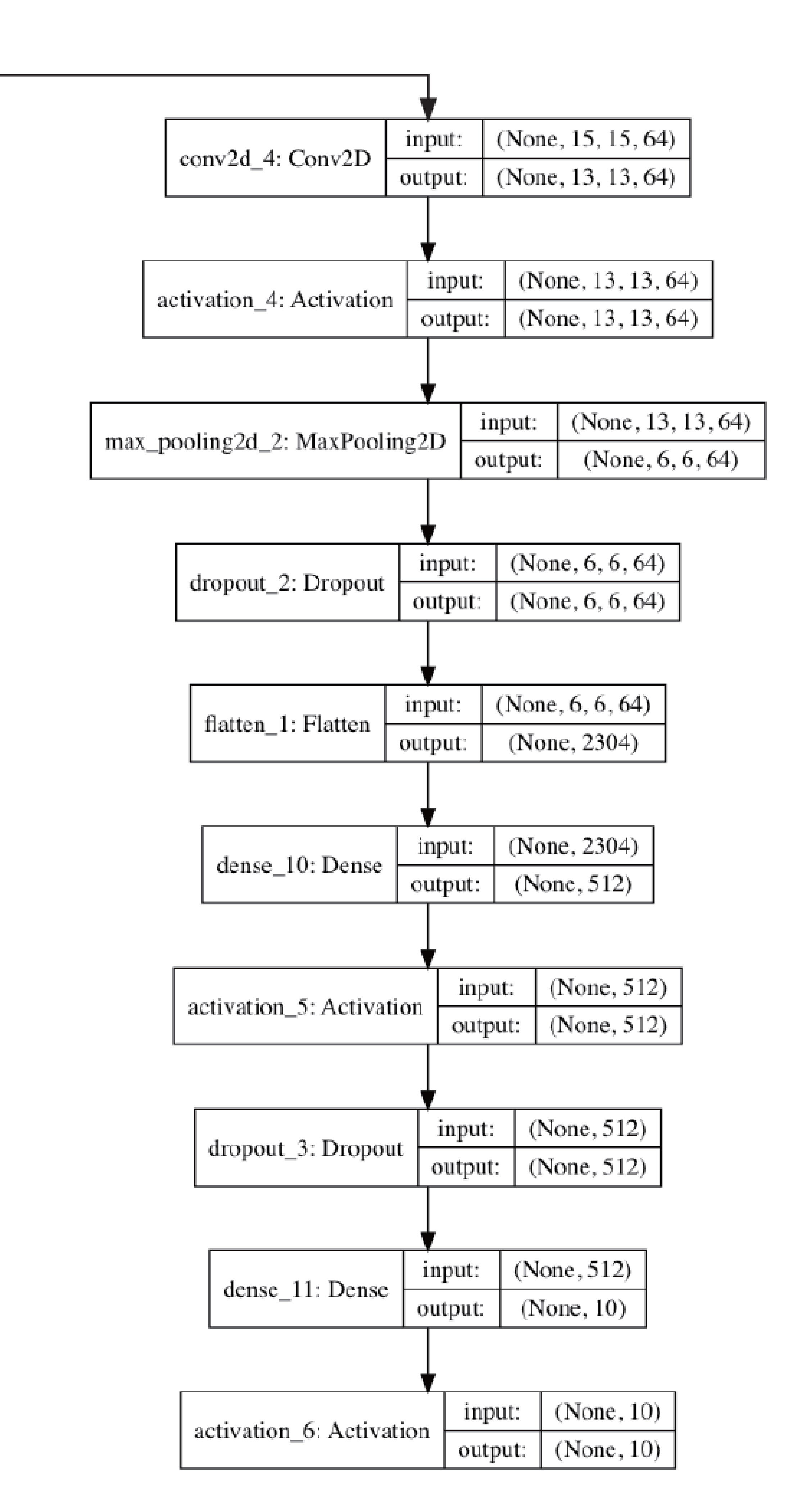


图 8-2 构建卷积神经网络用来分类 CIFAR-10 数据集

8.3 下游部分——使用凸优化模块训练模型

本节将使用凸优化模块训练模型：

```
    adam = Adam(lr=0.0001)
    model.compile(loss='categorical_crossentropy',
                       optimizer=adam,
                       metrics=['accuracy'])
```

最后，开始训练模型，并且评估模型准确性：

```
    # 训练模型
    best_model = ModelCheckpoint("cifar10_best.h5", monitor='val_loss',
verbose=0, save_best_only=True)
    tb = TensorBoard(log_dir="./logs")
    model.fit_generator(
                datagen.flow(X_train, Y_train, batch_size=batch_size),
                steps_per_epoch=X_train.shape[0] // batch_size,
                epochs=nb_epoch, verbose=1,
                validation_data=(X_test, Y_test),
                callbacks=[best_model,tb])

    # 模型评分
    score = model.evaluate(X_test, Y_test, verbose=0)
    # 输出结果
    print('Test score:', score[0])
    print("Accuracy: %.2f%%" % (score[1]*100))
    print("Compiled!")
```

以上代码作者使用 GPU 测试执行，50 个 epoch 中，每个 epoch 用时 12 秒左右，总计用时在 15 分钟以内。约 25 个 epoch 后，验证集的准确率数会逐步收敛在 0.8 左右，最终的准确率是 82.53%。

这并不是一个很高的准确率，又应当如何进一步提高模型的分类准确率呢？让我们开始本书实战阶段的内容吧，通过实战代码的讲解来谈一谈如何构建一个更好的模型。

8.4 参考文献及网页链接

[1] CIFAR-10 - Object Recognition in Images | Kaggle Available at: https://www.kaggle.com/c/cifar-10/data.

[2] Fchollet. fchollet/keras. GitHub Available at: https://github.com/fchollet/keras/blob/master/examples/cifar10_cnn.py.

第 9 章

见多识广——使用迁移学习提升准确率

本章将会介绍深度学习中非常实用的一个技术：迁移学习。本章的所有内容都会围绕一个任务，识别一张图是猫还是狗。9.1 节先会搭建一个普通的深度神经网络来尝试解决该问题，9.2 节会使用迁移学习技术，9.3 节会介绍模型融合技术，达到非常高的准确率。

接下来的内容，我们以 kaggle 上的一个比赛——猫狗大战（https://www.kaggle.com/c/ dogs-vs-cats-redux-kernels-edition）的数据集为例，来谈谈如何基于迁移学习实现高准确率的猫狗图像分类。该比赛提供了 25000 张图作为训练集，猫狗各占一半，测试集 12500 张，没有标定是猫还是狗。

我们先随机取一部分样本，进行可视化：

```
import os
import cv2
import random
import matplotlib.pyplot as plt

%matplotlib inline
```

```
%config InlineBackend.figure_format = 'retina'

path = 'train'
filenames = os.listdir(path)
plt.figure(figsize=(12, 10))
for i, filename in enumerate(random.sample(filenames, 12)):
    plt.subplot(3, 4, i+1)
    plt.imshow(cv2.imread(os.path.join(path, filename))[:,:,::-1])
   plt.title(filename)
```

其结果如图 9-1 所示。

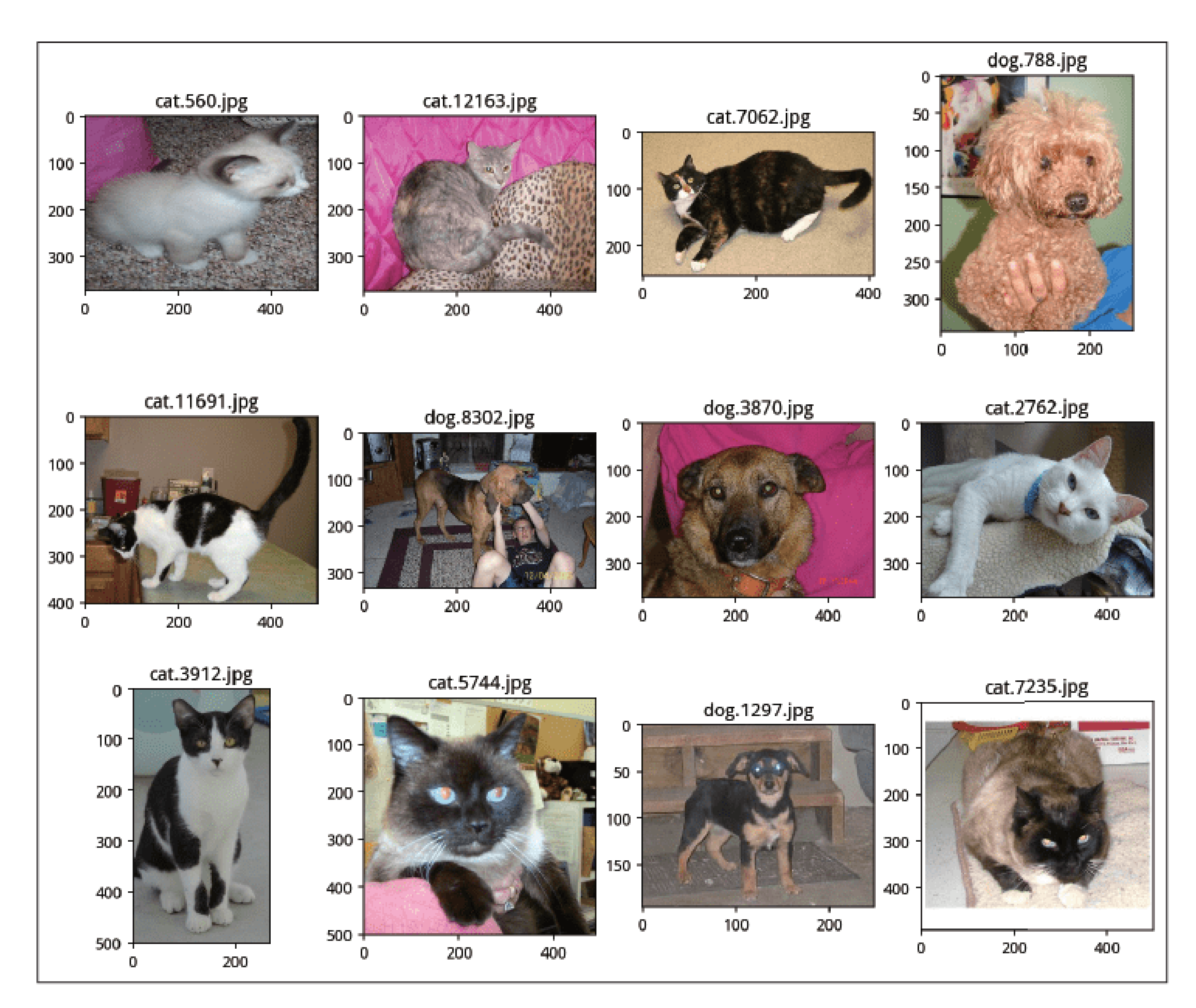


图 9-1　随机挑选出来的 12 张猫狗图像可视化，可以看到形状大小不一

其中，带百分号的那两行代码是 jupyter notebook 中特有的，它能够让 matplotlib 在网页中显示出比较好看的图片。

我们可以看到数据集图像大小不一，颜色多样。图像中的主体除了是猫狗以外，还有可能是人，因此识别起来并不简单。下面搭建一个卷积神经网络来尝试一下。

9.1　猫狗大战 1.0——使用卷积神经网络直接进行训练

9.1.1　导入数据

我们在 Linux 上可以通过 ls 命令和 head 命令很容易地查看数据集 train test 文件夹中的文件名:

```
ls train | head
```

运算结果：

```
# out
cat.0.jpg
cat.1.jpg
cat.10.jpg
cat.100.jpg
cat.1000.jpg
cat.10000.jpg
cat.10001.jpg
cat.10002.jpg
cat.10003.jpg
cat.10004.jpg
```

我们可以看到数据集的文件名都是由这样的形式命名的：cat. 序号 .jpg 和 dog. 序号 .jpg，猫狗各占一半，因此序号的范围是 0~12499。

由于数据的格式非常规则，因此我们很容易写出下面的代码：

```
import cv2
import numpy as np
from tqdm import tqdm

n = 25000
width = 128

X = np.zeros((n, width, width, 3), dtype=np.uint8)
y = np.zeros((n,), dtype=np.uint8)

for i in tqdm(range(n/2)):
    X[i] = cv2.resize(cv2.imread('train/cat.%d.jpg' % i), (width, 
width))
    X[i+n/2] = cv2.resize(cv2.imread('train/dog.%d.jpg' % i), (width, 
width))

y[n/2:] = 1
```

由于模型中存在全连接层，我们必须将所有图片都整理成一样的大小，因此我们先使用 cv2.imread 读取图片，然后用 cv2.resize 将图片缩放至一样的大小，这样就完成了数据的读取。

为了适配 Keras 的训练 API，我们先构建了一个（n, width, width, 3）的四维矩阵，然后一张一张图放入 X 的对应位置，而不是使用 Python 原生的数组。

9.1.2 可视化

为了确定读取的数据是正确的，随机取几张图看看：

```
import random
import matplotlib.pyplot as plt

%matplotlib inline
%config InlineBackend.figure_format = 'retina'

plt.figure(figsize=(12, 10))
for i in range(12):
    random_index = random.randint(0, n-1)
    plt.subplot(3, 4, i+1)
    plt.imshow(X[random_index][:,:,::-1])
    plt.title(['cat', 'dog'][y[random_index]])
```

其结果如图 9-2 所示。



图 9-2　随机挑选出来的 12 张调整过形状的图片

我们可以看到猫狗的图片已经全部变成（128, 128）形状了。

9.1.3 分割训练集和验证集

我们知道，如果没有验证集，就无法评估模型的性能、无法调参、无法知道什么时候应该停止训练，因此需要分离一部分数据做验证集。

这里不需要分离测试集，因为 kaggle 本身就已经提供了没有 label 的测试集。

```
    from sklearn.model_selection import train_test_split
    X_train, X_valid, y_train, y_valid = train_test_split(X, y, test_
size=0.2)
```

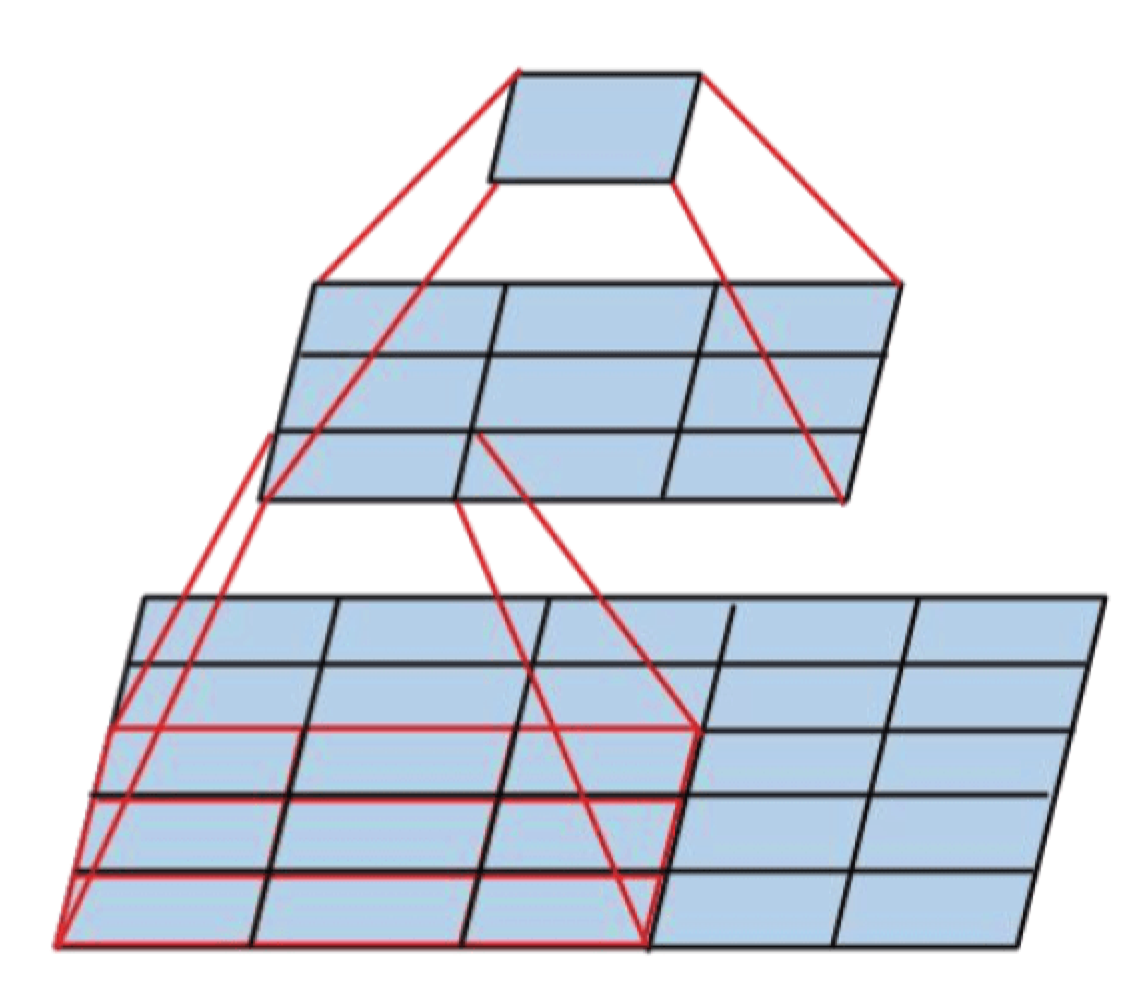
代码很简单，使用了 sklearn 的 train_test_split 函数。

分离以后的 X_train 有 20000 个样本，X_valid 有 5000 个样本，顺序都已经被打乱了。

在搭建模型之前，让我们来介绍一下 VGG16 模型。

VGG16 是一个很经典的模型，它的特征提取部分只使用了 3×3 的卷积核，以及 2×2 的池化层。在它之前很多人都认为卷积核要比较大才能识别更大的区域，不过根据计算可以知道，两个卷积核大小为 3×3 的卷积层可以有效覆盖 5×5 的区域，同时还可以减少计算量，以及增加非线性能力，这也是 VGG 系列模型的核心思想，就是减小卷积核，加深网络，如图 9-3 所示。

VGG 的名字来自于 Visual Geometry Group，这是牛津大学的一个实验室。他们发的论文标题也很直接：Very Deep Convolutional Networks for Large-Scale Visual Recognition，意思是用于大规模视觉识别的非常深的卷积神经网络。



（图来自论文 Rethinking the Inception Architecture for Computer Vision）

图 9-3 两层 3×3 卷积层覆盖区域可视化

因此，最终 VGG16 的完整结构是这样的：

```
img_input = Input(shape=input_shape)
# Block 1
x = Conv2D(64, (3, 3), activation='relu', padding='same',
         name='block1_conv1')(img_input)
x = Conv2D(64, (3, 3), activation='relu', padding='same',
         name='block1_conv2')(x)
x = MaxPooling2D((2, 2), strides=(2, 2), name='block1_pool')(x)

# Block 2
x = Conv2D(128, (3, 3), activation='relu', padding='same',
         name='block2_conv1')(x)
x = Conv2D(128, (3, 3), activation='relu', padding='same',
         name='block2_conv2')(x)
x = MaxPooling2D((2, 2), strides=(2, 2), name='block2_pool')(x)

# Block 3
x = Conv2D(256, (3, 3), activation='relu', padding='same',
         name='block3_conv1')(x)
x = Conv2D(256, (3, 3), activation='relu', padding='same',
         name='block3_conv2')(x)
x = Conv2D(256, (3, 3), activation='relu', padding='same',
         name='block3_conv3')(x)
x = MaxPooling2D((2, 2), strides=(2, 2), name='block3_pool')(x)

# Block 4
x = Conv2D(512, (3, 3), activation='relu', padding='same',
         name='block4_conv1')(x)
x = Conv2D(512, (3, 3), activation='relu', padding='same',
         name='block4_conv2')(x)
x = Conv2D(512, (3, 3), activation='relu', padding='same',
         name='block4_conv3')(x)
x = MaxPooling2D((2, 2), strides=(2, 2), name='block4_pool')(x)

# Block 5
x = Conv2D(512, (3, 3), activation='relu', padding='same',
         name='block5_conv1')(x)
x = Conv2D(512, (3, 3), activation='relu', padding='same',
         name='block5_conv2')(x)
x = Conv2D(512, (3, 3), activation='relu', padding='same',
         name='block5_conv3')(x)
x = MaxPooling2D((2, 2), strides=(2, 2), name='block5_pool')(x)

# Classification block
```

```
    x = Flatten(name='flatten')(x)
    x = Dense(4096, activation='relu', name='fc1')(x)
    x = Dense(4096, activation='relu', name='fc2')(x)
    x = Dense(classes, activation='softmax', name='predictions')(x)

    model = Model(inputs, x, name='vgg16')
```

参考链接：https://github.com/fchollet/keras/blob/master/keras/applications/vgg16.py。

我们在第 11 章部分还会介绍如何基于 VGG 模型构建更复杂的全卷积神经网络，用于图像分割。

9.1.4 搭建模型

由于我们的数据集只有 25000 张图，而 ImageNet 有上百万张图，我们的问题规模比较小，因此在搭建模型的时候进行了一些修改：

- 卷积核个数从 32 开始按照 2 的 n 次方递增，比如 32、64、128……
- 卷积层后面添加 BatchNormalization 层加速训练。
- 去除巨大的全连接隐藏层，采用 GlobalAveragePooling2D 降维。

由于我们的问题是二分类问题，因此分类器使用 sigmoid 激活函数，修改一个神经元。

代码如下：

```
from keras.layers import *
from keras.models import *

inputs = Input((width, width, 3))
x = inputs
for i, layer_num in enumerate([2, 2, 3, 3, 3]):
    for j in range(layer_num):
        x = Conv2D(32*2**i, 3, padding='same', activation='relu')(x)
        x = BatchNormalization()(x)
        x = Activation('relu')(x)
    x = MaxPooling2D(2)(x)

x = GlobalAveragePooling2D()(x)
x = Dropout(0.5)(x)
x = Dense(1, activation='sigmoid')(x)

model = Model(inputs, x)
```

搭建好模型以后，让我们来指定优化器和 loss：

```
model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

如 7.3 节提到的，对于 CNN 分类问题而言，直接使用 Adam 优化器以及对应的默认学习率（lr=0.001）来测试模型是一种快速判断模型是否收敛的“套路”。由于我们的问题是二分类问题，激活函数是 sigmoid，所以 loss 选择 binary_crossentropy。如果是多分类问题，那么有多少类就设置多少个神经元，然后使用 softmax 作为激活函数，使用 categorical_crossentropy 作为 loss。

9.1.5 模型训练

模型训练的代码很简单：

```
h = model.fit(X_train, y_train, batch_size=128, epochs=20,
         validation_data=(X_valid, y_valid))
```

代码中的 h 是为了保存模型训练过程中每一代的状态，比如 loss、准确率等。

我们选择 128 作为 batch_size，这个是经验值，如果你没有几百个 GPU，一般是越大越好。我们总共训练了 20 代，最后验证集大概能达到 90% 以上的准确率。

为了直观地感受模型训练的效果，可以用下面的代码画出历史曲线：

```
plt.figure(figsize=(10, 4))
plt.subplot(1, 2, 1)
plt.plot(h.history['loss'])
plt.plot(h.history['val_loss'])
plt.legend(['loss', 'val_loss'])
plt.ylabel('loss')
plt.xlabel('epoch')

plt.subplot(1, 2, 2)
plt.plot(h.history['acc'])
plt.plot(h.history['val_acc'])
plt.legend(['acc', 'val_acc'])
plt.ylabel('acc')
plt.xlabel('epoch')
```

其结果如图 9-4 所示。

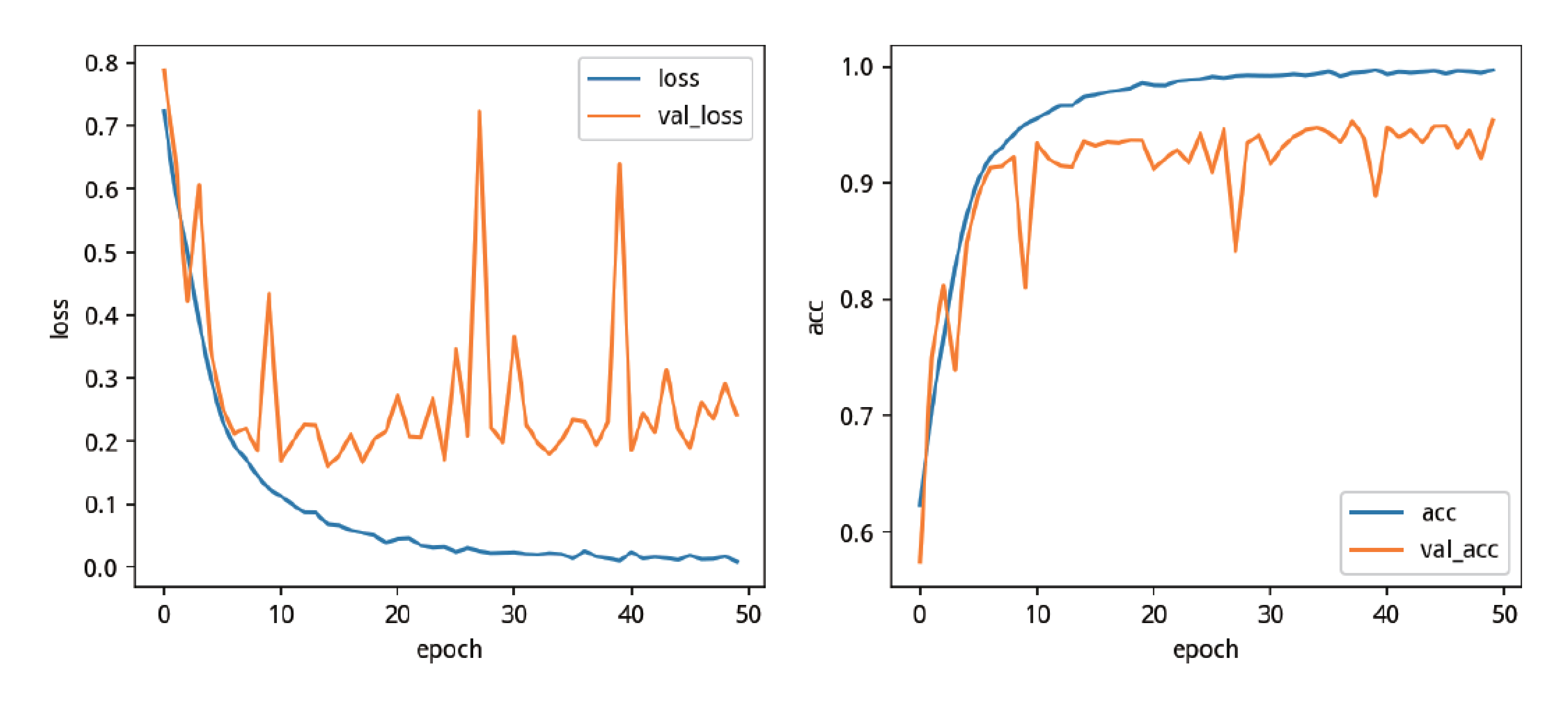


图 9-4 卷积神经网络训练过程的 loss 和 acc 可视化

模型在验证集上最高准确率是 0.9224。

9.1.6 总结

由于这个模型在验证集上的准确率都不够高，就不用浪费机会提交到 kaggle 官网去评测模型在测试集上的表现了。

9.2 猫狗大战 2.0——使用 ImageNet 数据集预训练模型

9.2.1 迁移学习

迁移学习是一个很有意义的技术，它能够直接利用一个身经百战的模型脑子里的知识来学习新的数据集，达到与之前相当、甚至比之前的模型还好的表现。

在没有迁移学习以前，人们训练一个模型通常要非常久的时间。例如，如果有 4 块 NVIDIA Titan Black 显卡，想在 ImageNet 数据集上训练一个 VGG16 模型，那么进行完整的训练需要 2~3 周的时间。这是非常浪费时间、浪费电费的做法，完全可以利用别人已经训练好的 VGG16 模型的权值，然后利用其中的卷积核权重。

在 ImageNet 数据集上预训练过的权重，靠近输入的那几层卷积层一般都是识别各种边缘信息或者颜色信息，除非是与 ImageNet 差异非常大的数据集，通常情况下训练出来权重可视化都类似图 9-5。



图 9-5

如果搭建的模型以这些权值初始化，不管新的任务需要识别的东西有没有在 ImageNet 的类别中，最后训练效果一定会比使用随机初始化强得多，所以试试使用迁移学习来做猫狗大战吧。

ImageNet

ImageNet 是李飞飞发起的一个图像数据集，就 VGG 参加的 ILSVRC 2014 来说，该比赛数据集约有 120 万张图片，归属于 1000 个类，每个图都是高清大图。ImageNet 为图像技术带来了重要的革命，有了大规模的图像数据集，卷积神经网络变得更加强大。在 2015 年甚至已经超越了人类水平。

ImageNet: A Large-Scale Hierarchical Image Database Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification

9.2.2 数据预处理

VGG 论文的第 3.1 节中提到的预处理步骤是这样的：

During training, the input to our ConvNets is a fixed-size 224 × 224 RGB image. The only preprocessing we do is subtracting the mean RGB value, computed on the training set, from each pixel.

意思是说，图片需要裁剪为（224, 224）的大小，预处理方法是减去训练集每个像素点的颜色的平均值。

因此，首先在载入数据的时候，我们需要把宽度修改为 224： width = 224。然后可以写出这样的预处理函数：

```
def preprocess_input(x):
    return x - [103.939, 116.779, 123.68]
```

9.2.3 搭建模型

好了，终于可以利用经过 ImageNet 训练的 VGG16 模型了。

首先通过 VGG16 获取一个 cnn_model：

```
from keras.applications import VGG16

cnn_model = VGG16(include_top=False,
                  input_shape=(width, width, 3),
                  weights='imagenet')
for layer in cnn_model.layers:
layer.trainable = False
```

这里将 cnn_model 的每一层都给锁住了，因为不希望模型最开始几个识别基本几何特征的卷积层被改变。

接下来利用 preprocess_input 预处理图片，再用 cnn_model 提取特征，最后搭建分类器：

```
inputs = Input((width, width, 3))
x = inputs
x = Lambda(preprocess_input, name='preprocessing')(x)
x = cnn_model(x)
x = GlobalAveragePooling2D()(x)
x = Dense(512, activation='relu')(x)
x = Dropout(0.5)(x)
x = Dense(1, activation='sigmoid')(x)

model = Model(inputs, x)
model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

9.2.4 模型可视化

为了直观地感受模型结构，可以用下面的代码进行可视化（见图 9-6）：

```
from IPython.display import SVG
from keras.utils.vis_utils import model_to_dot

SVG(model_to_dot(model, show_shapes=True).create(prog='dot',
format='svg'))
```

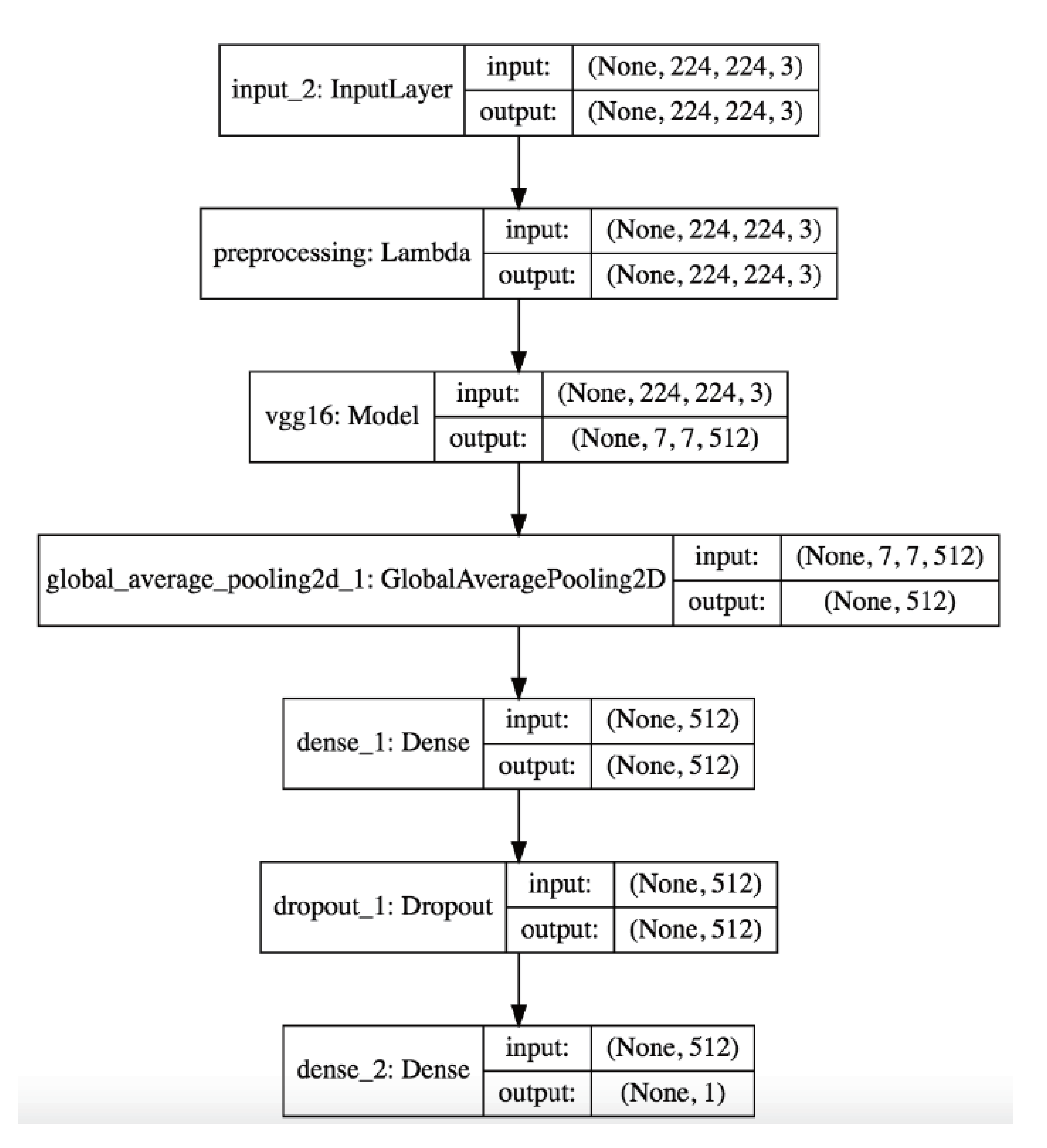


图 9-6　VGG 模型可视化

9.2.5　训练模型

训练代码与上面一样，但是曲线明显好看许多，如图 9-7 所示。

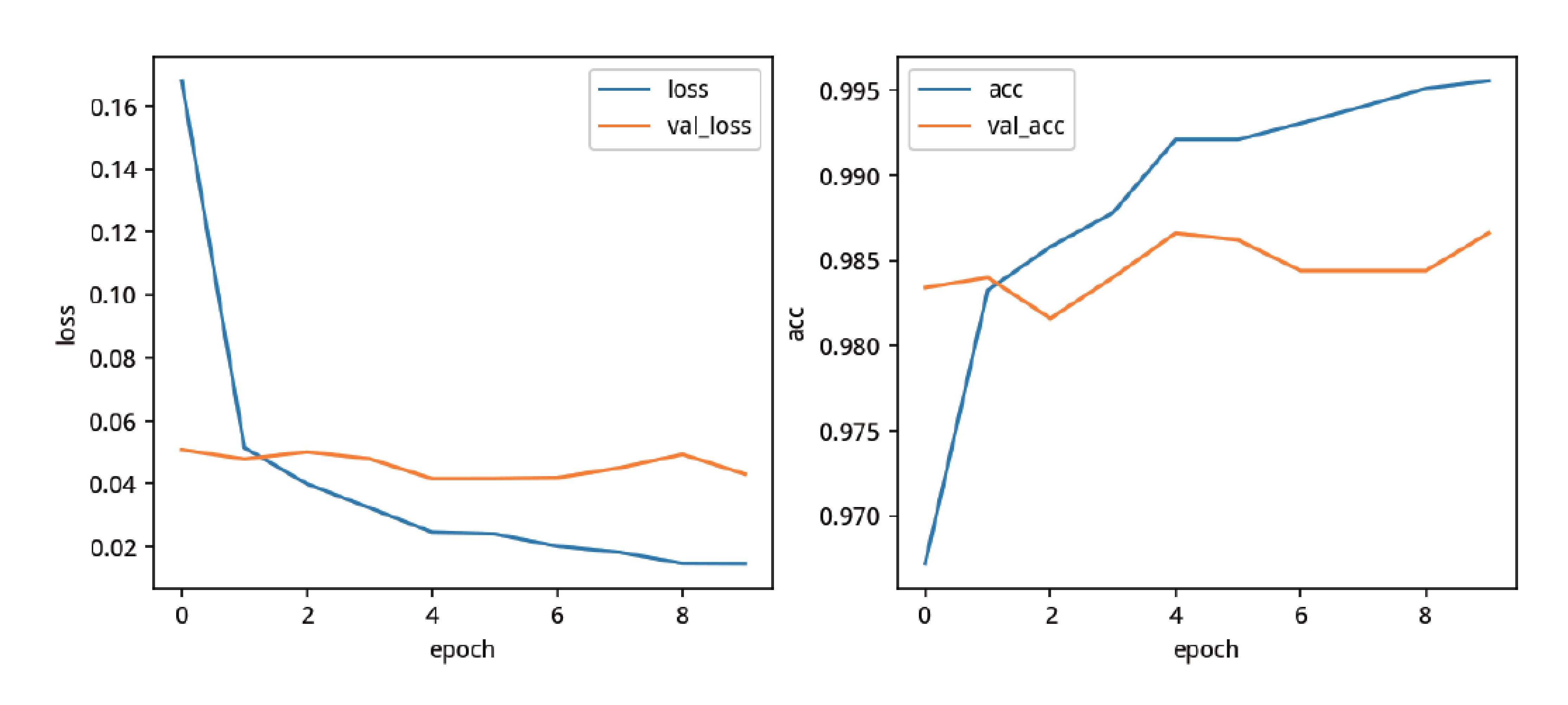


图 9-7　VGG 模型训练过程的 loss 和 acc 可视化

模型在验证集上最高准确率是 0.9866。

9.2.6 提交到 kaggle 评估

首先读取测试集，代码很简单，就不多解释了：

```
n = 12500
X_test = np.zeros((n, width, width, 3), dtype=np.uint8)

for i in tqdm(range(n)):
    X_test[i] = cv2.resize(cv2.imread('test/%d.jpg' % (i+1)), (width, 
width))
```

然后用训练好的模型进行预测：

```
y_pred = model.predict(X_test, batch_size=128, verbose=1)
```

之后导入提交模板，将我们的输出裁剪到（0.005, 0.995）的范围内，最后输出到 csv 文件：

```
import pandas as pd

df = pd.read_csv('sample_submission.csv')
df['label'] = y_pred.clip(min=0.005, max=0.995)
df.to_csv('pred.csv', index=None)
```

提交到 kaggle 后，我们可以得到 0.06241 左右的成绩，位于 134/1314，也就是 10% 左右了。

9.3 猫狗大战 3.0——使用多种预训练模型组合提升表现

我们知道，迁移学习是不需要修改前面几十层卷积层的，但是在训练的时候依然会浪费很多时间在 cnn_model 上，这是不必要的。为了节省时间，可以先用 cnn_model 预测得到特征，再合并特征，训练分类器进行分类即可。这种基于原模型直接得到预测的特征，并基于原模型预测特征做进一步分析的方法称为**快速迁移学习**。

快速迁移学习的步骤如下：

- 载入数据集
- 搭建预训练模型（cnn_model）
- 导出数据集的特征
- 搭建简单全连接分类器模型（model）
- 训练模型

下面进行详细讲解。

9.3.1 载入数据集

载入数据集还是和上面的代码一样，只是 Inception V3 和 Xception 需要把 width 改为 299。

```
import cv2
import numpy as np
from tqdm import tqdm

n = 25000
width = 224

X = np.zeros((n, width, width, 3), dtype=np.uint8)
y = np.zeros((n,), dtype=np.uint8)

for i in tqdm(range(n/2)):
    X[i] = cv2.resize(cv2.imread('train/cat.%d.jpg' % i), (width,
width))
    X[i+n/2] = cv2.resize(cv2.imread('train/dog.%d.jpg' % i), (width, width))

y[n/2:] = 1
```

9.3.2 使用正确的预处理函数

首先载入一些必要的库：

```
from keras.layers import *
from keras.models import *
from keras.applications import *
from keras.optimizers import *
```

接下来，如果是 VGG16 / ResNet50 模型，就用这个预处理函数：

```
def preprocess_input(x):
    return x - [103.939, 116.779, 123.68]
```

如果是 Inception V3 / Xception 模型，就用这个预处理函数：

```
from keras.applications.inception_v3 import preprocess_input
```

当然也可以用这个：

```
from keras.applications.xception import preprocess_input
```

9.3.3 搭建特征提取模型并导出特征

cnn_model 的结构很简单，如图 9-8 所示。

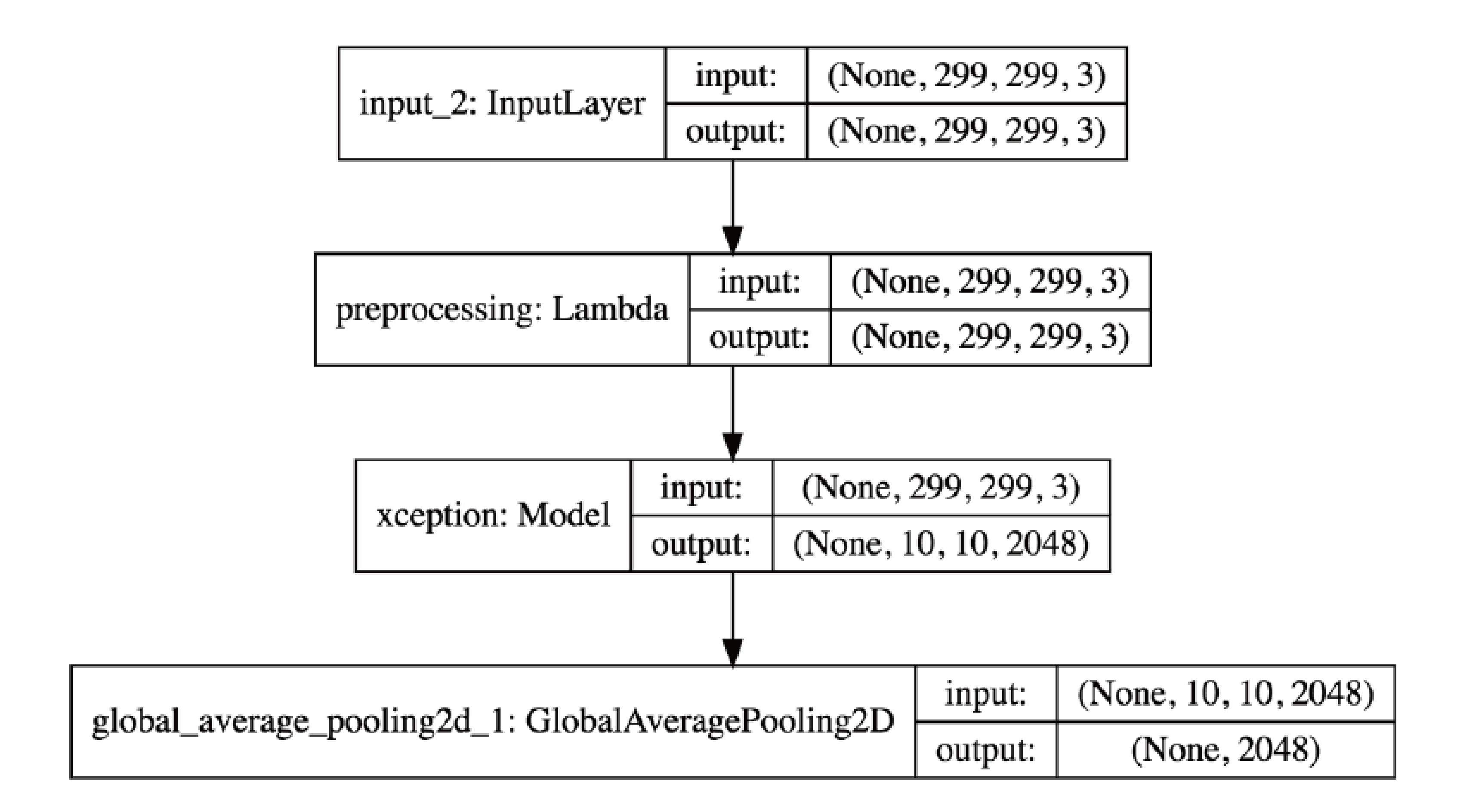


图 9-8 特征提取模型可视化

先进行对应的预处理，然后是模型，最后一个 GLobalAveragePooling2D 把特征图转为特征向量。Xception 的代码如下：

```
    cnn_model = Xception(include_top=False, input_shape=(width, width, 3),
weights='imagenet')

    inputs = Input((width, width, 3))
    x = inputs
    x = Lambda(preprocess_input, name='preprocessing')(x)
    x = cnn_model(x)
    x = GlobalAveragePooling2D()(x)
    cnn_model = Model(inputs, x)

    features = cnn_model.predict(X, batch_size=128, verbose=1)
```

如果要用 Inception V3 或者 ResNet50，只需要修改成对应的模型即可。

9.3.4 搭建并训练全连接分类器模型

有了特征以后，训练分类器就非常快了，可以说是一秒一代。分类器的结构也很简单，一个 Dropout 防止过拟合，然后接一个全连接分类器就好了，如图 9-9 所示。

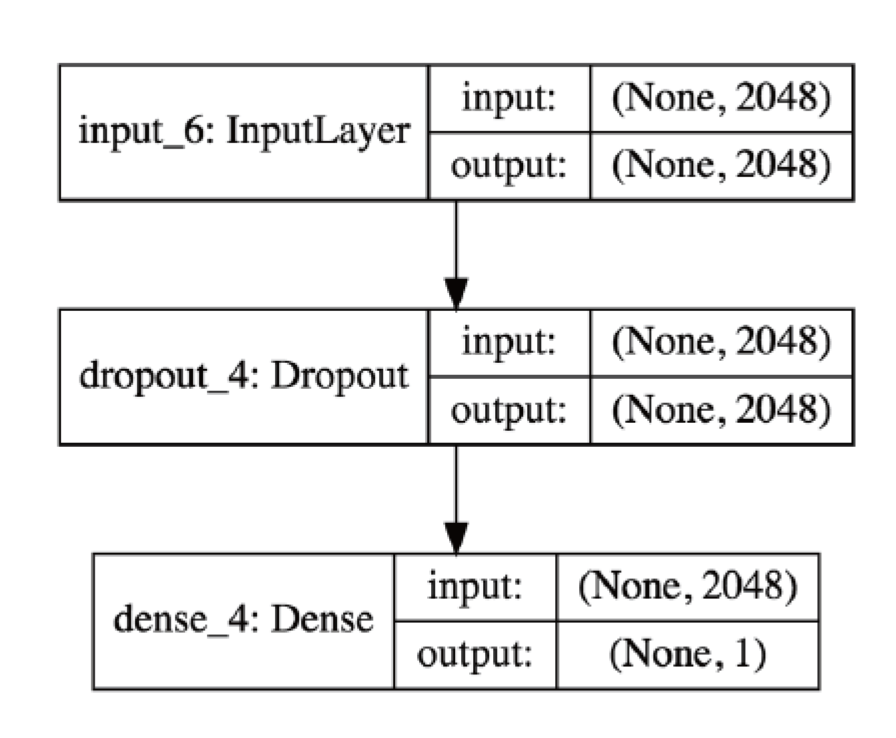


图 9-9 分类器模型可视化

代码如下：

```
inputs = Input(features.shape[1:])
x = inputs
x = Dropout(0.5)(x)
x = Dense(1, activation='sigmoid')(x)
model = Model(inputs, x)
model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])
h = model.fit(features, y, batch_size=128, epochs=10, validation_split=0.2)
```

这里使用了 validation_split 来自动切分 20% 的验证集。

Xception 的训练曲线如图 9-10 所示。

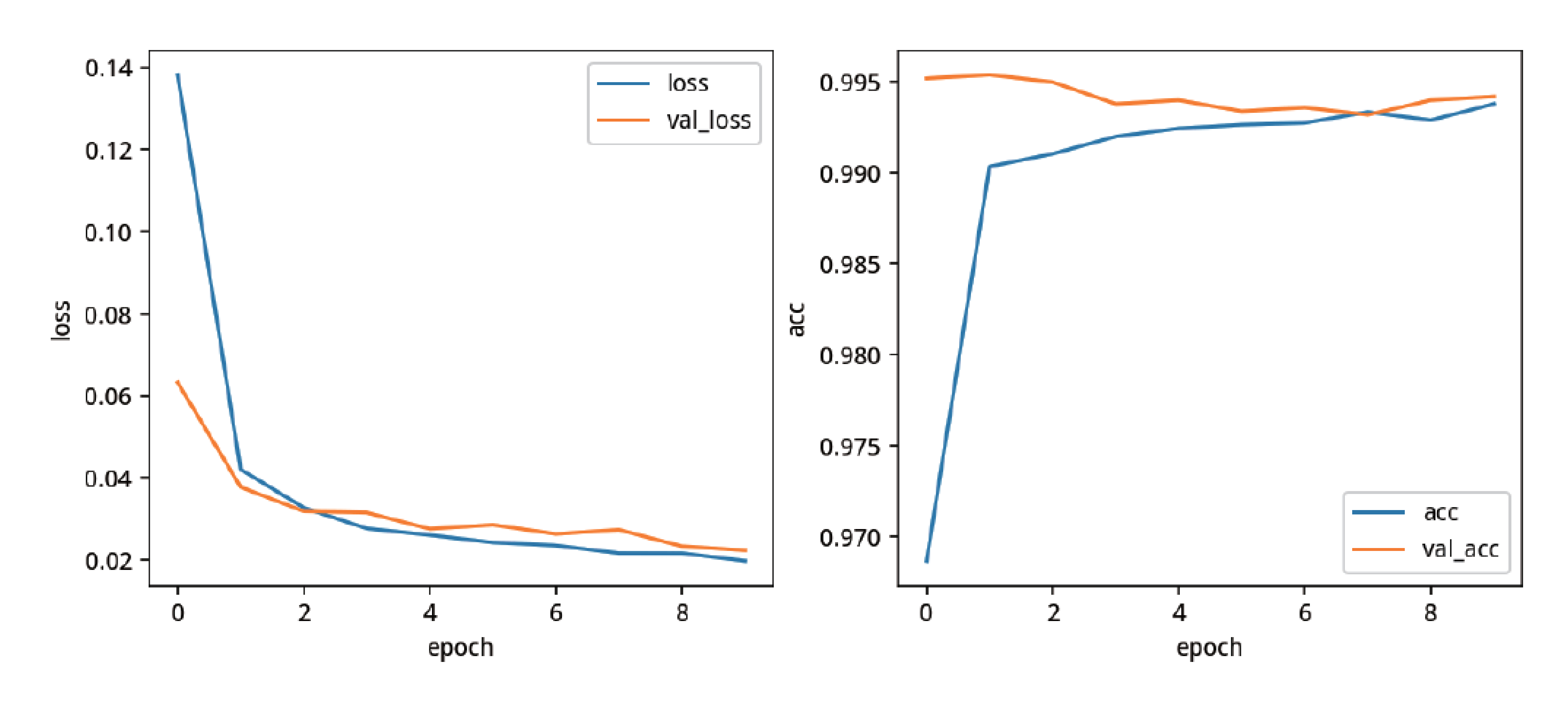


图 9-10 Xception 训练过程 loss 和 acc 可视化

其他的模型也大同小异。

9.3.5　在测试集上预测

首先导入测试集数据：

```
n = 12500
X_test = np.zeros((n, width, width, 3), dtype=np.uint8)

for i in tqdm(range(n)):
    X_test[i] = cv2.resize(cv2.imread('test/%d.jpg' % (i+1)), (width, width))
```

然后计算测试集特征：

```
features_test = cnn_model.predict(X_test, batch_size=128, verbose=1)
```

利用模型预测测试集是猫还是狗：

```
y_pred = model.predict(features_test, batch_size=128)
```

输出到 csv 文件中：

```
import pandas as pd

df = pd.read_csv('sample_submission.csv')
df['label'] = y_pred.clip(min=0.005, max=0.995)
df.to_csv('pred.csv', index=None)
```

9.4 融合模型

我们对各个模型都进行迁移学习，并且将预测结果提交到 kaggle，结果如下：

模　型	分　数
VGG16	0.06241
ResNet50	0.05045
Inception V3	0.04516
Xception	0.04469

从上面的结果可以看到，Xception 和 Inception V3 的效果是比较好的，因此可以先融合这两个模型。

融合模型的方法很简单，首先将特征提取出来，然后拼接在一起，构建一个全连接分类器训练就可以了。

模型融合能够提高成绩的理论依据是，有些模型辨认猫准确率高，有些模型辨认狗准确率高，给这些模型不同的权重，让它们能够取长补短，综合各自的优势。为了能够更好地融合模型，可以提取特征进行融合，这样会有更好的效果，弱特征的权重会越学越小，强特征会越学越大，最后得到效果非常好的模型。

9.4.1 获取特征

为了方便获取特征，可以写出这样的函数：

```
def get_features(MODEL, data=X):
    cnn_model = MODEL(include_top=False, input_shape=(width, width, 3),
weights='imagenet')

    inputs = Input((width, width, 3))
```

```
        x = inputs
        x = Lambda(preprocess_input, name='preprocessing')(x)
        x = cnn_model(x)
        x = GlobalAveragePooling2D()(x)
        cnn_model = Model(inputs, x)

        features = cnn_model.predict(data, batch_size=64, verbose=1)
        return features
```

这样就可以简化代码，很方便地获取各个模型对应的特征。例如：

```
    inception_features = get_features(InceptionV3, X)
    xception_features = get_features(Xception, X)
    features = np.concatenate([inception_features, xception_features],
axis=-1)
```

inception_features 的 shape 是（25000, 2048），xception_features 的 shape 也 是（25000, 2048），那么经过 np.concatenate 函数拼接，指定轴为最后一个维度以后，shape 就会变成（25000, 4096），可以理解为这是两个预训练模型对这些图片的理解。

9.4.2 数据持久化

如果不想每次导出特征，可以使用 h5py 将特征保存为 hdf5 格式的文件，以便下次使用：

```
    import h5py

    with h5py.File('features', 'w') as d:
        d['features'] = features
```

读入也很简单：

```
    import h5py

    with h5py.File('features', 'r') as d:
        features = np.array(d['features'])
```

这里使用 np.array 将数据转换为 numpy 矩阵，目的是将数据载入内存中，这样训练的速度会快一些。

9.4.3 构建模型

构建模型的代码和之前的全连接模型代码完全一样，这里就不贴了，结果如图 9-11 所示。

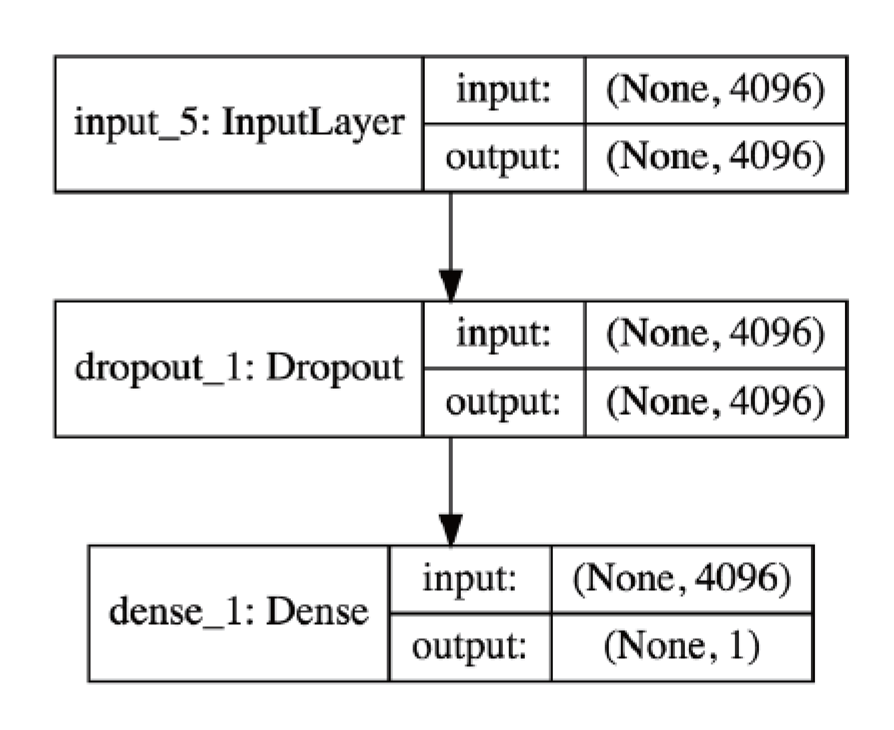


图 9-11 融合模型可视化

训练曲线如图 9-12 所示。

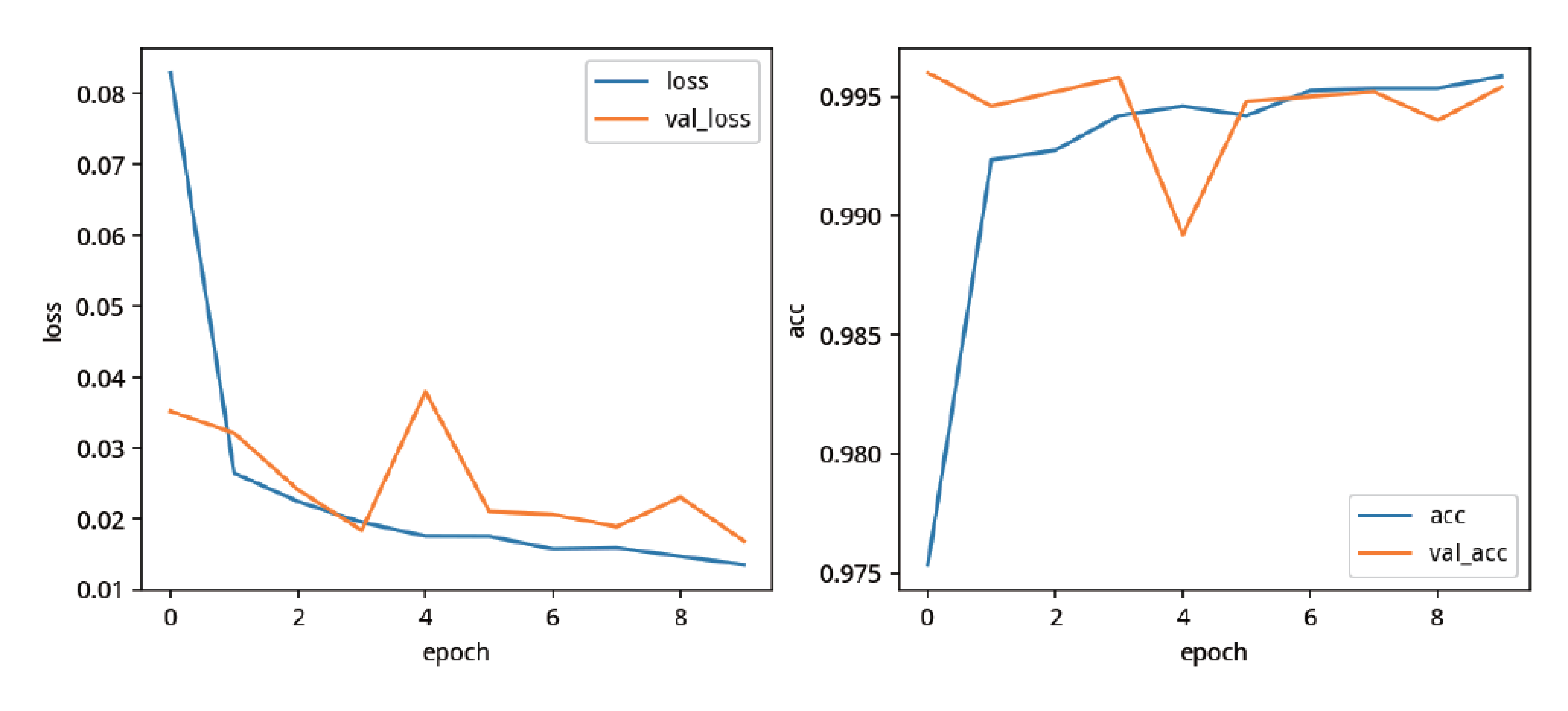


图 9-12 融合模型训练 loss 和 acc 可视化

9.4.4 在测试集上预测

在测试集上预测的过程和上面单模型差不多，只是输入数据也需要融合一下，下面是完整的代码：

```
n = 12500
X_test = np.zeros((n, width, width, 3), dtype=np.uint8)

for i in tqdm(range(n)):
    X_test[i] = cv2.resize(cv2.imread('test/%d.jpg' % (i+1)), (width,
width))

inception_features = get_features(InceptionV3, X_test)
xception_features = get_features(Xception, X_test)
features_test = np.concatenate([inception_features, xception_features],
axis=-1)
```

```
y_pred = model.predict(features_test, batch_size=128)

import pandas as pd

df = pd.read_csv('sample_submission.csv')
df['label'] = y_pred.clip(min=0.005, max=0.995)
df.to_csv('pred.csv', index=None)
```

9.5 总　结

这个模型在 kaggle 上面获得了 0.04077 的分数，排在 17/1314，大概是全球 1.2% 的水平。如果还想继续提升成绩，可以采取下面几种方式：

- 使用更好的预训练模型导出特征
- 对预训练模型进行微调（fine-tune）
- 使用数据增强生成更多数据
- 对数据集进行清洗，去除异常值
- 使用更多正则化方法防止过拟合

异常值样图如图 9-13 所示。

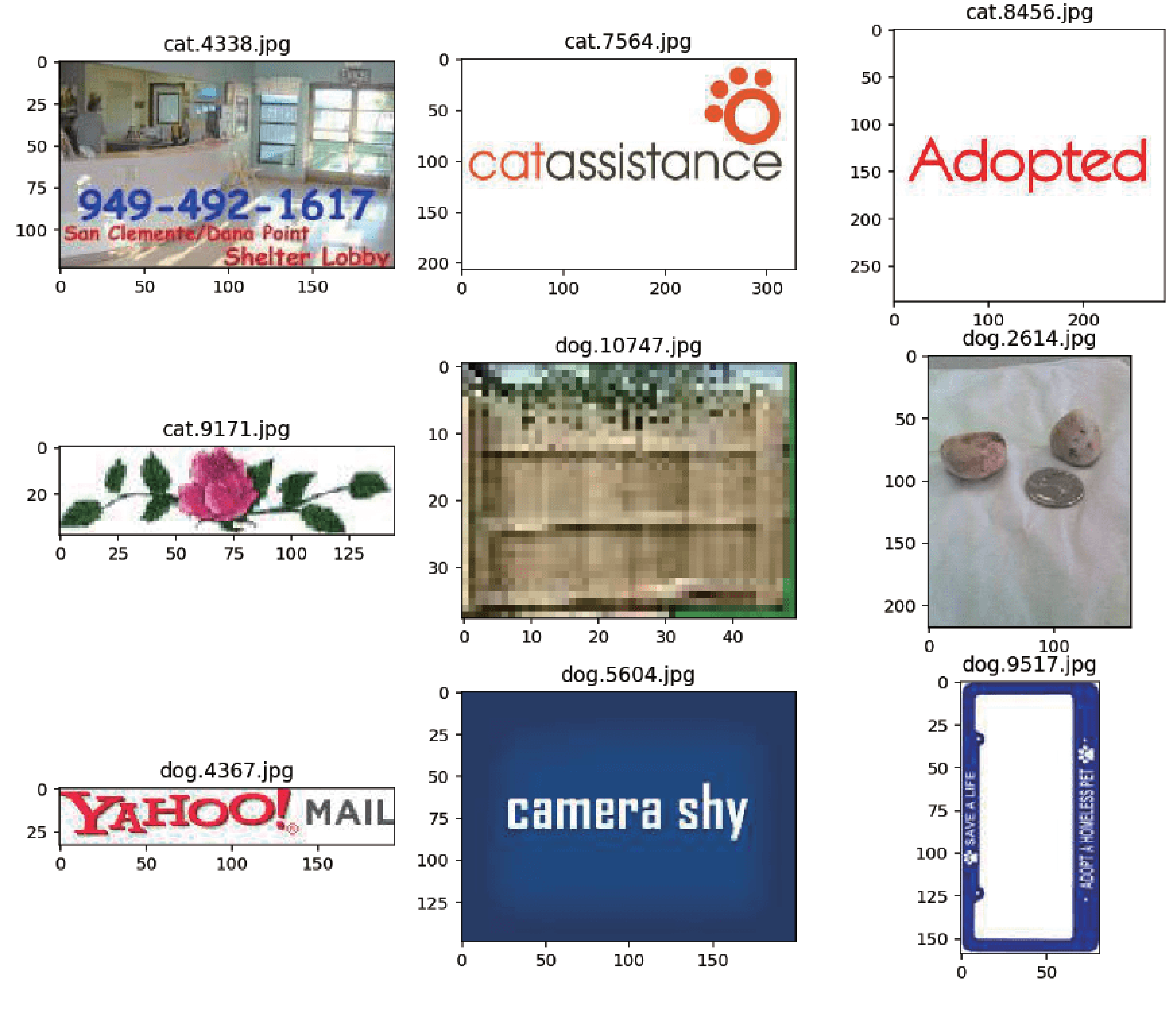


图 9-13　异常图片抽样可视化

9.6 参考文献及网页链接

［1］ Dogs vs. Cats Redux: Kernels Edition | Kaggle. Available at: https://www.kaggle.com/c/dogs-vs-cats-redux-kernels-edition.

［2］ CS231n Convolutional Neural Networks for Visual Recognition. Available at: http://cs231n.github.io/convolutional-networks/.

［3］ Chollet, F. Xception: Deep Learning with Depthwise Separable Convolutions. [1610.02357] Xception: Deep Learning with Depthwise Separable Convolutions (2017). Available at: https://arxiv.org/ abs/1610.02357.

［4］ Fchollet. fchollet/keras. GitHub. Available at: https://github.com/fchollet/keras/blob/master/keras/applications/vgg16.py.

［5］ He, K., Zhang, X., Ren, S. & Sun, J. Deep Residual Learning for Image Recognition. [1512.03385] Deep Residual Learning for Image Recognition (2015). Available at: https://arxiv.org/abs/1512.03385.

［6］ ImageNet. ImageNet Available at: http://image-net.org/.

［7］ Simonyan, K. & Zisserman, A. Very Deep Convolutional Networks for Large-Scale Image Recognition. [1409.1556] Very Deep Convolutional Networks for Large-Scale Image Recognition (2015). Available at: https://arxiv.org/abs/1409.1556.

［8］ Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J. & Wojna, Z. Rethinking the Inception Architecture for Computer Vision. [1512.00567] Rethinking the Inception Architecture for Computer Vision (2015). Available at: https://arxiv.org/abs/1512.00567.

第 10 章

看图识字——使用深度神经网络进行文字识别

本章将介绍如何识别一张图片中的文字。首先使用卷积神经网络加多个全连接分类器的方法来识别，然后使用卷积神经网络结合循环神经网络的方式来识别，不仅能够准确识别字符，不需要进行切割，还能够根据上下文以及语法规则猜测字符。

10.1 使用卷积神经网络进行端到端学习

本节通过 Keras 搭建一个深度卷积神经网络来识别 captcha 验证码。由于深度卷积神经网络训练需要很大的计算量，因此建议使用显卡来运行本项目。

首先介绍本节需要用到的一个验证码生成库—— captcha。

captcha

captcha 是一个生成验证码的 Python 库，它支持图片验证码和语音验证码，我们使用的是它生成图片验证码的功能。该项目的地址是：https://github.com/lepture/captcha。

安装方法如下：

```
pip install captcha
```

我们的实验环境（见本书 1.4 节）中已经集成了 captcha 0.2.2，这里无须安装。

我们设置的验证码格式为数字加大写字母，生成一串验证码试试看：

```
from captcha.image import ImageCaptcha
import matplotlib.pyplot as plt
import numpy as np
import random

%matplotlib inline
%config InlineBackend.figure_format = 'retina'

import string
characters = string.digits + string.ascii_uppercase
print(characters)

width, height, n_len, n_class = 170, 80, 4, len(characters)

generator = ImageCaptcha(width=width, height=height)
random_str = ''.join([random.choice(characters) for j in range(4)])
img = generator.generate_image(random_str)

plt.imshow(img)
plt.title(random_str)
```

其结果如图 10-1 所示。

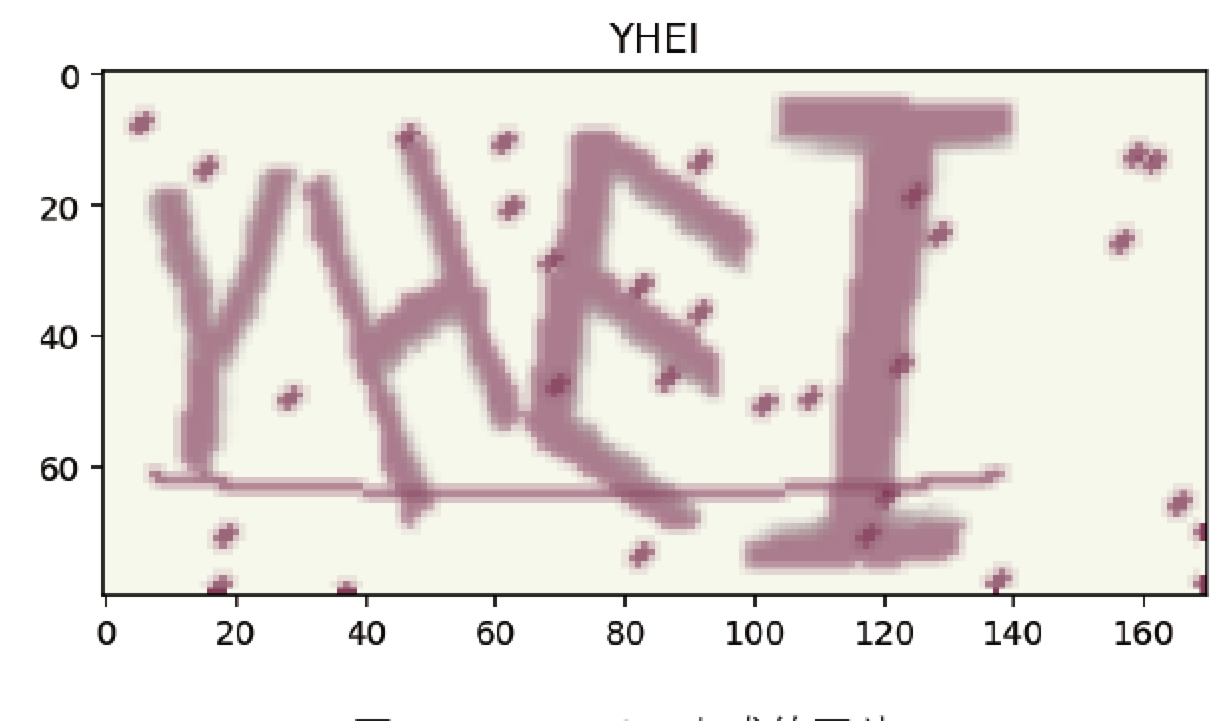


图 10-1 captcha 生成的图片

10.1.1 编写数据生成器

如第 6 章内容强调的一样，在训练模型的过程中，可以选择两种方式来生成训练数据：一种是一次性生成几万张图，然后开始训练；另一种是定义一个数据生成器，然后利用 fit_generator 函数进行训练，以类似愚公移山的方式处理海量输入训练数据。

第一种方式的好处是训练时显卡利用率高，如果需要经常调参，可以一次生成，然后多次训练；第二种方式的好处是不需要生成大量数据，训练过程中可以利用 CPU 生成数据，而且还有一个好处是可以无限生成数据，提高模型的泛化能力。

我们需要生成的数据是 X 和 y，X 是一批图片，y 是对应的数据标签，也就是希望模型识别的结果。

1. 图片矩阵 X

X 的形状是（batch_size, height, width, 3），例如一批生成 32 个样本，图片宽度为 170，高度为 80，那么形状就是（32, 80, 170, 3），取第一张图就是 X[0]。

2. 标签矩阵 y

y 的形状是 4 个（batch_size, n_class），如果转换成 numpy 的格式，则是（n_len, batch_size, n_class），例如一批生成 32 个样本，验证码的字符有 36 种，长度是 4 位，那么它的形状就是 4 个（32, 36），也可以说是（4, 32, 36）。

```
def gen(batch_size=32):
    X = np.zeros((batch_size, height, width, 3), dtype=np.uint8)
    y = [np.zeros((batch_size, n_class), dtype=np.uint8) for i in
range(n_len)]
    generator = ImageCaptcha(width=width, height=height)
    while True:
        for i in range(batch_size):
            random_str = ''.join([random.choice(characters) for j in
range(4)])
            X[i] = generator.generate_image(random_str)
            for j, ch in enumerate(random_str):
                y[j][i, :] = 0
                y[j][i, characters.find(ch)] = 1
        yield X, y
```

上面就是一个可以无限生成数据的例子，下面将使用这个生成器来训练模型。

10.1.2 使用生成器

生成器的使用方法很简单，只需要使用 Python 内置的 next 函数即可。下面是一个例子，一批生成 32 个数据，然后显示第一个数据的图片。由于 label 使用了 One-Hot 编码，所以还需要

对 label 进行解码，首先将它转为 numpy 数组，然后取 36 个数字中最大数字的位置，实际上这些数字就是对应字符的概率，因此取最大的就行，最后将概率最大的 4 个字符转换为字符串。

```
def decode(y, index=0):
    y = np.argmax(np.array(y), axis=2)[:, index]
    return ''.join([characters[x] for x in y])

X, y = next(gen(1))
plt.imshow(X[0])
plt.title(decode(y))
```

10.1.3 构建深度卷积神经网络

```
from keras.models import *
from keras.layers import *

input_tensor = Input((height, width, 3))
x = input_tensor
for i in range(4):
    x = Convolution2D(32*2**i, 3, activation='relu')(x)
    x = Convolution2D(32*2**i, 3, activation='relu')(x)
    x = MaxPooling2D(2)(x)

x = Flatten()(x)
x = Dropout(0.25)(x)
x = [Dense(n_class, activation='softmax', name='c%d'%(i+1))(x) for i in
range(4)]
model = Model(inputs=input_tensor, outputs=x)

model.compile(loss='categorical_crossentropy',
              optimizer='adadelta',
              metrics=['accuracy'])
```

模型结构很简单，特征提取部分使用的是两个卷积，一个池化的结构，该结构是学习的 VGG16 的结构。之后将输出的特征 Flatten 为一维向量，然后添加 Dropout，尽量避免过拟合问题，最后连接 4 个分类器，每个分类器是 36 个神经元，输出 36 个字符的概率。

10.1.4 模型可视化

得益于 Keras 自带的可视化，可以使用几句代码来可视化模型的结构（见图 10-2）。

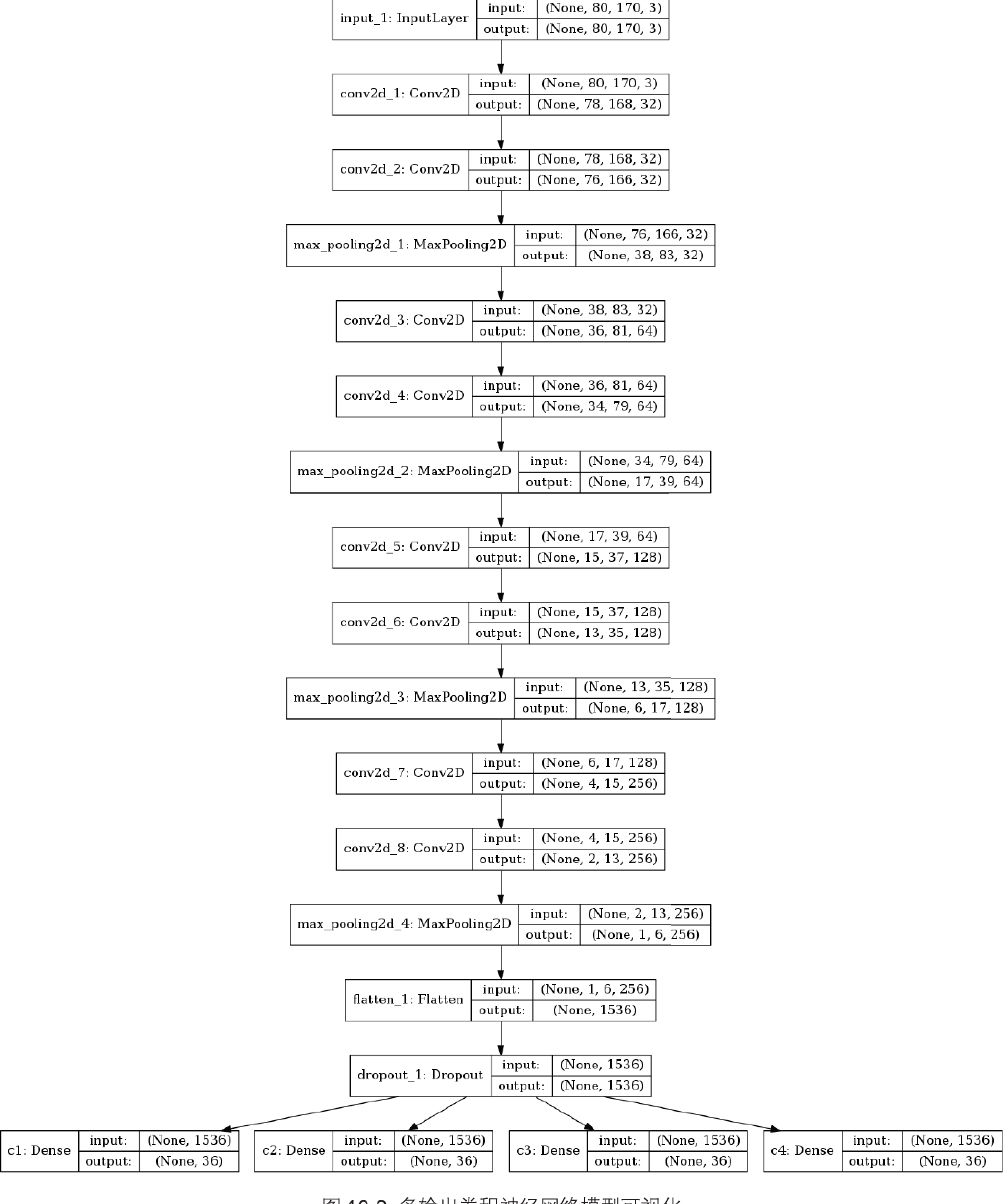


图 10-2　多输出卷积神经网络模型可视化

```
from keras.utils.visualize_util import plot
from IPython.display import Image

plot(model, to_file="model.png", show_shapes=True)
Image('model.png')
```

这里需要使用 pydot 库和 graphviz 库，在第 1 章介绍的基于 NVIDIA Docker 搭建的环境中已经安装完毕，可以直接使用。

我们可以看到最后一层卷积层输出的形状是（1, 6, 256），已经不能再加卷积层了。

10.1.5 训练模型

训练模型是所有步骤里面最简单的一个，直接使用 model.fit_generator 函数即可，这里的验证集使用了同样的生成器，由于数据是通过生成器随机生成的，因此数据重复的概率接近于 0。注意，这段代码在笔记本上可能要耗费一下午时间。如果你想让模型预测得更准确，可以将 epochs 改得更大，但它也将耗费成倍的时间。注意，这里使用了一个小技巧，添加 workers =2 参数让 Keras 自动实现多进程生成数据，摆脱 Python 单线程效率低的缺点。

模型训练每代会使用 128*400=51200 个样本，验证的时候会使用 128*10=1280 个样本。

```
h = model.fit_generator(gen(128), steps_per_epoch=400, epochs=20,
                        workers=4, pickle_safe=True,
                        validation_data=gen(128), validation_steps=10)
```

训练 20 代以后，可以画出训练过程的 loss 和 acc 曲线图（见图 10-3）：

```
plt.figure(figsize=(10, 4))
plt.subplot(1, 2, 1)
plt.plot(h.history['loss'])
plt.plot(h.history['val_loss'])
plt.legend(['loss', 'val_loss'])
plt.ylabel('loss')
plt.xlabel('epoch')
plt.ylim(0, 1)

plt.subplot(1, 2, 2)
for i in range(4):
    plt.plot(h.history['val_c%d_acc' % (i+1)])

plt.legend(['val_c%d_acc' % (i+1) for i in range(4)])
plt.ylabel('acc')
plt.xlabel('epoch')
plt.ylim(0.9, 1)
```

注意，这里使用 plt.ylim 限制了 y 的范围，不然范围会很大，无法看出具体趋势。

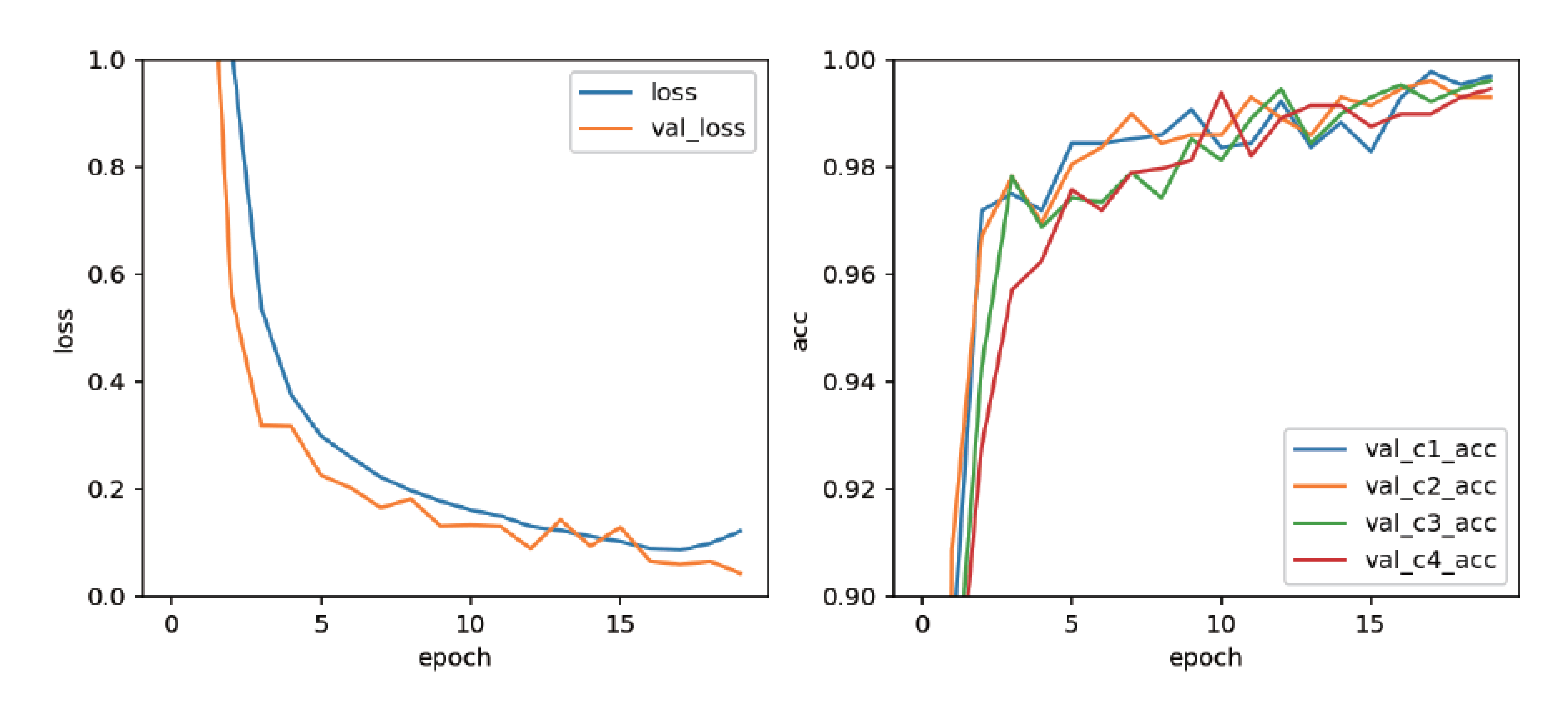


图 10-3　多输出卷积神经网络模型 loss 可视化以及每个分类器的 acc 可视化

10.1.6 计算模型总体准确率

模型在训练的时候只会显示每一个字符的准确率，为了统计模型的总体准确率，可以编写下面的函数：

```
from tqdm import tqdm

def evaluate(batch_size=128, steps=20):
    batch_acc = 0
    generator = gen(batch_size)
    for i in tqdm(range(steps)):
        X, y = next(generator)
        y_pred = model.predict(X)
        y_pred = np.argmax(y_pred, axis=-1)
        y_true = np.argmax(y, axis=-1)
        batch_acc += np.equal(y_true, y_pred).all(axis=0).mean()
    return batch_acc / steps

evaluate(model)
```

这里用到了一个 tqdm 库，它是一个进度条的库，目的是能够实时反馈进度。接下来，通过一些 numpy 计算去统计准确率，这里计算规则是只要发现一个错，就不算它对。经过计算，模型的总体准确率经过 20 代训练就可以达到 97.8%，继续训练还可以达到更高的准确率。

10.1.7 测试模型

训练完成之后，可以识别几个验证码试试看（见图 10-4）：

```
X, y = next(gen(12))
y_pred = model.predict(X)
```

```
plt.figure(figsize=(16, 8))
for i in range(12):
    plt.subplot(3, 4, i+1)
    plt.title('real: %s\npred:%s'%(decode(y, i), decode(y_pred, i)))
    plt.imshow(X[i])
```

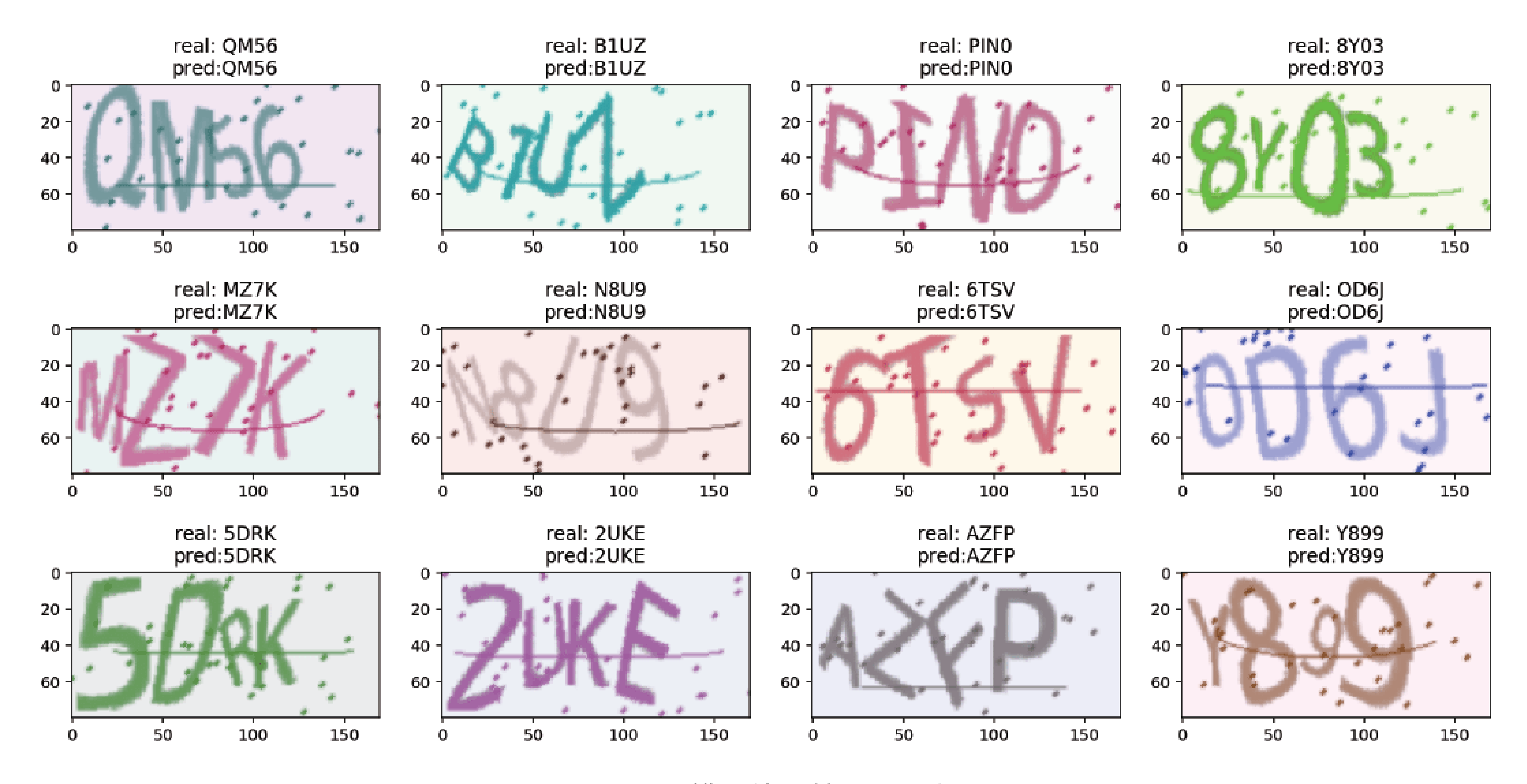


图 10-4 模型输出结果可视化

10.1.8 模型总结

模型的大小是 16MB，在笔者的笔记本上跑 1000 张验证码需要用 20 秒，当然有显卡的话会更快。对于验证码识别的问题来说，哪怕是 10% 的准确率也已经称得上破解，毕竟假设 100% 识别率破解要一个小时，那么 10% 的识别率也只用十个小时，还算等得起，而识别率达到 97%，已经可以称得上完全破解了这类验证码。模型大小完全不可能存下这么多的验证码图片，说明它是真的需要去辨认这些验证码，而不只是按像素去比对。

10.2 使用循环神经网络改进模型

对于这种按照顺序书写的文字，还有一种方法可以使用，那就是循环神经网络来识别序列。下面了解一下如何使用循环神经网络来识别这类验证码。这部分的代码和上面一样，只是将 n_class 改为 len(characters)+1，因为需要添加一个空白类用于 CTC Loss。

```
from captcha.image import ImageCaptcha
import matplotlib.pyplot as plt
import numpy as np
```

```
import random

%matplotlib inline
%config InlineBackend.figure_format = 'retina'

import string
characters = string.digits + string.ascii_uppercase
print(characters)

width, height, n_len, n_class = 170, 80, 4, len(characters)+1

generator = ImageCaptcha(width=width, height=height)
random_str = ''.join([random.choice(characters) for j in range(4)])
img = generator.generate_image(random_str)

plt.imshow(img)
plt.title(random_str)
```

10.2.1 CTC Loss

这个 loss 是一个特别神奇的 loss，它可以在只知道序列的顺序、不知道具体位置的情况下让模型收敛（warp-ctc），如图 10-5 所示。

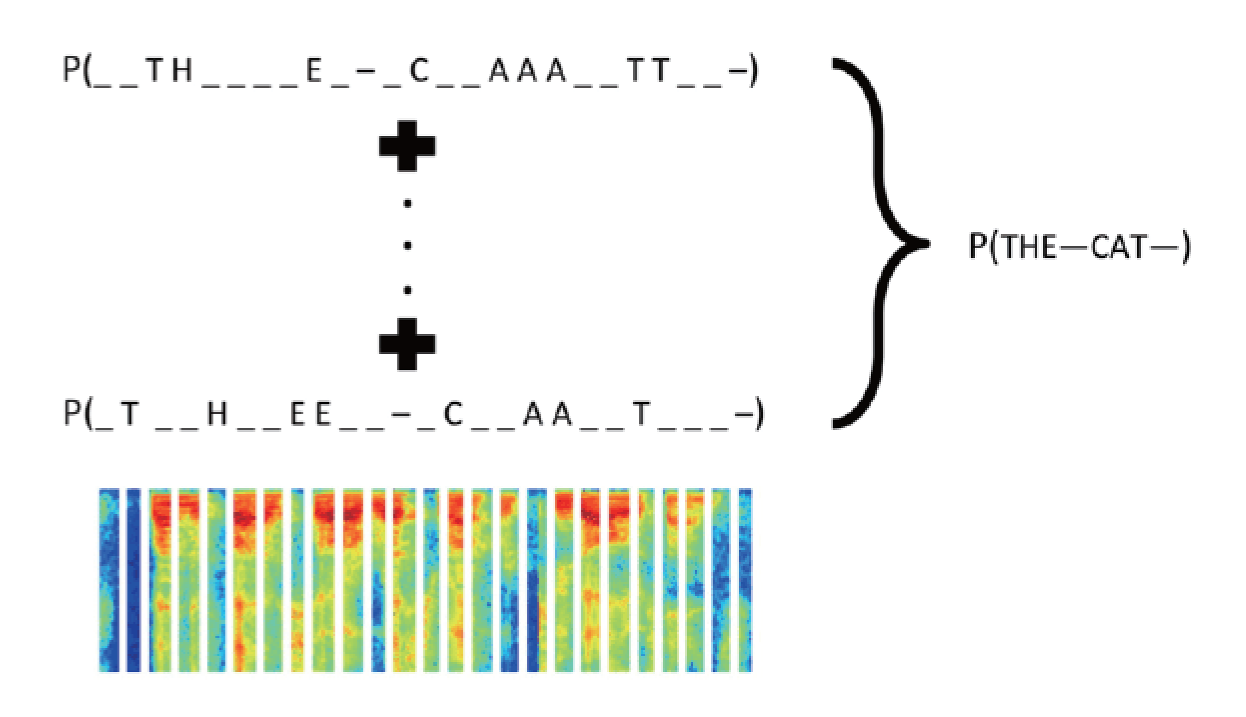


图 10-5　不同顺序的序列对应同一个 label 示意图

在 Keras 中，CTC Loss 已经内置了，直接定义这样一个函数即可。由于使用的是循环神经网络，因此默认丢掉前面两个输出，它们通常无意义，并且会影响模型的输出。

- y_pred 是模型的输出，是按照顺序输出的 37 个字符的概率，因为这里用到了循环神经网络，所以需要一个空白字符的类。
- labels 是验证码，是四个数字，每个数字对应字符的编号。
- input_length 表示 y_pred 的长度，这里是 15。
- label_length 表示 labels 的长度，这里是 4。

```
from keras import backend as K

def ctc_lambda_func(args):
```

```
    y_pred, labels, input_length, label_length = args
    y_pred = y_pred[:, 2:, :]
    return K.ctc_batch_cost(labels, y_pred, input_length, label_length)
```

10.2.2 模型结构

我们的模型结构是这样设计的，首先通过卷积神经网络去识别特征，然后经过一个全连接降维，再按照水平顺序输入到一种特殊的循环神经网络，叫 GRU（Gated Recurrent Unit），可以理解为是 LSTM 的简化版。LSTM 早在 1997 年就已经被发明出来了，但是 GRU 直到 2014 年才出现。经过实验，GRU 效果比 LSTM 要好。

参考链接：https://zhuanlan.zhihu.com/p/28297161

```
from keras.models import *
from keras.layers import *
from keras.optimizers import *
rnn_size = 128

input_tensor = Input((width, height, 3))
x = input_tensor
x = Lambda(lambda x:(x-127.5)/127.5)(x)
for i in range(3):
    for j in range(2):
        x = Convolution2D(32*2**i, 3, kernel_initializer='he_uniform')(x)
        x = BatchNormalization()(x)
        x = Activation('relu')(x)
    x = MaxPooling2D((2, 2))(x)

conv_shape = x.get_shape().as_list()
rnn_length = conv_shape[1]
rnn_dimen = conv_shape[2]*conv_shape[3]
print(conv_shape, rnn_length, rnn_dimen)
x = Reshape(target_shape=(rnn_length, rnn_dimen))(x)
rnn_length -= 2

x = Dense(rnn_size, kernel_initializer='he_uniform')(x)
x = BatchNormalization()(x)
x = Activation('relu')(x)
x = Dropout(0.2)(x)

gru_1 = GRU(rnn_size, return_sequences=True,
         kernel_initializer='he_uniform', name='gru1')(x)
gru_1b = GRU(rnn_size, return_sequences=True,
         kernel_initializer='he_uniform',
                go_backwards=True, name='gru1_b')(x)
```

```
    x = add([gru_1, gru_1b])
    gru_2 = GRU(rnn_size, return_sequences=True,
             kernel_initializer='he_uniform', name='gru2')(x)
    gru_2b = GRU(rnn_size, return_sequences=True,
             kernel_initializer='he_uniform',
                    go_backwards=True, name='gru2_b')(x)
    x = concatenate([gru_2, gru_2b])
    x = Dropout(0.2)(x)
    x = Dense(n_class, activation='softmax')(x)
    base_model = Model(inputs=input_tensor, outputs=x)
    labels = Input(name='the_labels', shape=[n_len], dtype='float32')
    input_length = Input(name='input_length', shape=[1], dtype='int64')
    label_length = Input(name='label_length', shape=[1], dtype='int64')
    loss_out = Lambda(ctc_lambda_func,
                    output_shape=(1,),
                    name='ctc'
       )([x, labels, input_length, label_length])
    model = Model(inputs=[input_tensor, labels, input_length, label_length],
outputs=[loss_out])
    model.compile(
       loss={'ctc': lambda y_true, y_pred: y_pred},
       optimizer='adam'
    )
```

从 Input 到最后一个 MaxPooling2D，是一个很深的卷积神经网络，它负责学习字符的各个特征，尽可能区分不同的字符。它输出 shape 是 [None, 17, 6, 128]，这个形状相当于把一张宽为 170、高为 80 的彩色图像（170, 80, 3）压缩为宽为 17、高为 6 的 128 维特征的特征图（17, 6, 128）。然后我们把图像 reshape 为（17, 768），也就是把高和特征放在一个维度，再降维成（17, 128），也就是从左到右有 17 条特征，每个特征 128 个维度。

这 128 个维度就是一条图像的非常高维、非常抽象的概括，然后将 17 个特征向量依次输入到 GRU 中，GRU 有能力学会不同特征向量的组合会代表什么字符，即使是字符之间有粘连也不害怕。这里使用了双向 GRU，最后 Dropout 接一个全连接层，作为分类器输出每个字符的概率。

这个是 base_model 的结构，也是我们模型的结构，那么后面的 labels、input_length、label_length 和 loss_out 都是为了输入必要的数据来计算 CTC Loss 的。

10.2.3 模型可视化

可视化的代码同上，这里只贴图（见图 10-6）。

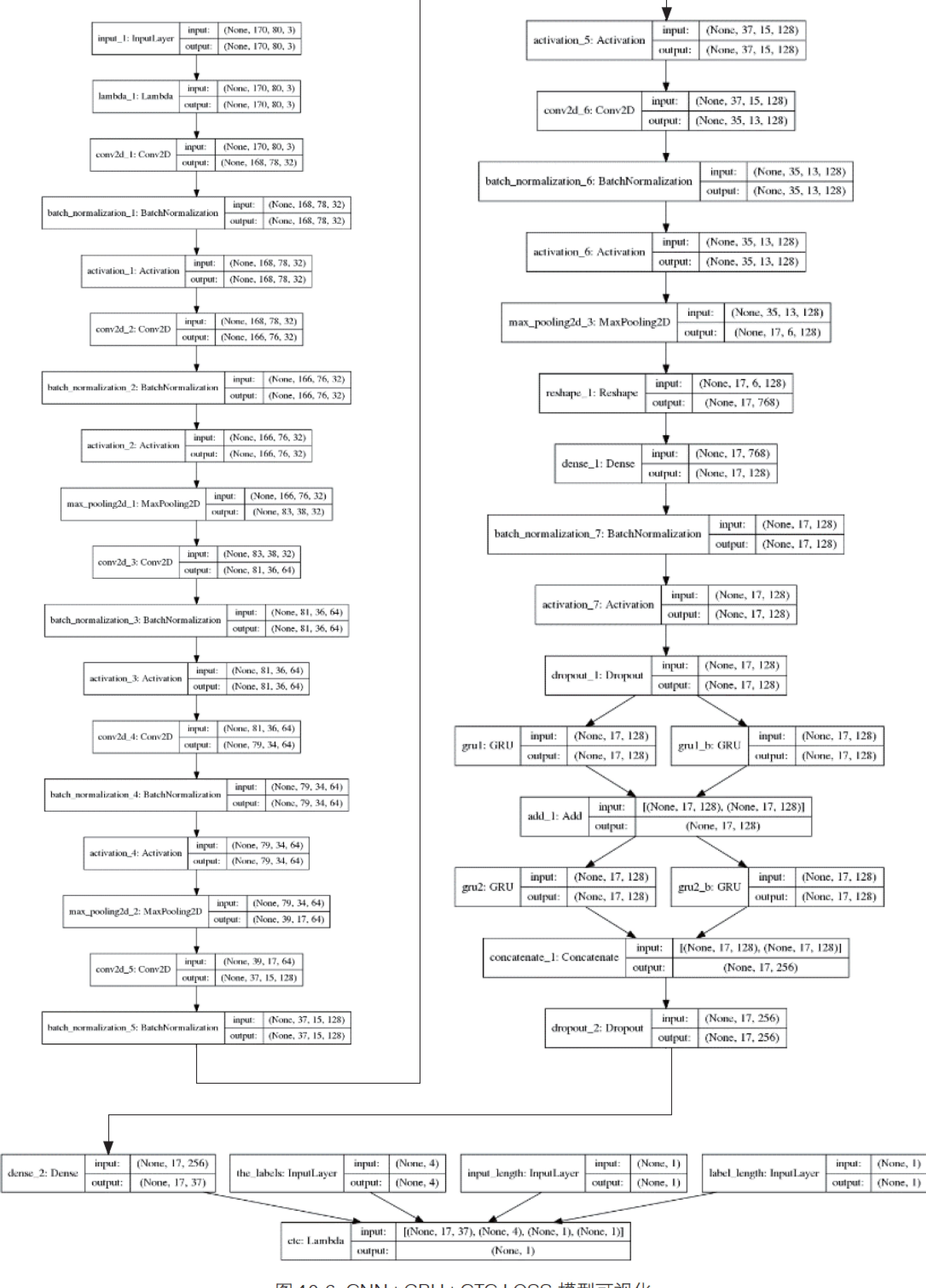


图 10-6 CNN+GRU+CTC LOSS 模型可视化

可以看到模型比上一个模型复杂了许多，但实际上只是因为输入比较多，所以它显得很大。还有一个值得注意的地方，我们的图片在输入时是经过了旋转的，这是因为希望以水平方向输入循环神经网络，而图片在 numpy 中默认是这样的形状（height, width, 3），因此使用了 transpose 函数将图片转为了（width, height, 3）的格式，这样就能够把 X 轴转到第一个维度，方便输入到循环神经网络。

10.2.4 数据生成器

根据模型的输入，需要输入 4 个数据：

- X 是一批图片。
- y 是每个图片对应的 label，最大长度为 n_len。
- input_length 表示模型输出的长度，这里是 15。
- label_length 表示 labels 的长度，这里是 4。

最后还有一个输入是 np.ones(batch_size)，这是因为 Keras 在训练模型的时候必须输入一个 X 和一个 y，这里把上面 4 个都合并为一个 X 了，因此实际上 y 没有参与 loss 的计算，所以随便编一个 batch_size 长度的数据输入进去就好了。

```
def gen(batch_size=128):
    X = np.zeros((batch_size, width, height, 3), dtype=np.uint8)
    y = np.zeros((batch_size, n_len), dtype=np.uint8)
    generator = ImageCaptcha(width=width, height=height)
    while True:
        for i in range(batch_size):
            random_str = ''.join([random.choice(characters)
                                  for j in range(n_len)])
            X[i] = np.array(
                generator.generate_image(random_str)
             ).transpose(1, 0, 2)
            y[i] = [characters.find(x) for x in random_str]
        yield [X, y, np.ones(batch_size)*rnn_length,
         np.ones(batch_size)*n_len], np.ones(batch_size)
```

可以举个例子，使用一次生成器，查看输出的是什么内容（见图 10-7）：

```
(X_vis, y_vis, input_length_vis, label_length_vis), _ = next(gen(1))
print(X_vis.shape, y_vis, input_length_vis, label_length_vis)

plt.imshow(X_vis[0].transpose(1, 0, 2))
plt.title(''.join([characters[i] for i in y_vis[0]]))
```

可以看到输出了下面的内容：

```
(1, 170, 80, 3) [[29 4 21 21]] [ 15.] [ 4.]
```

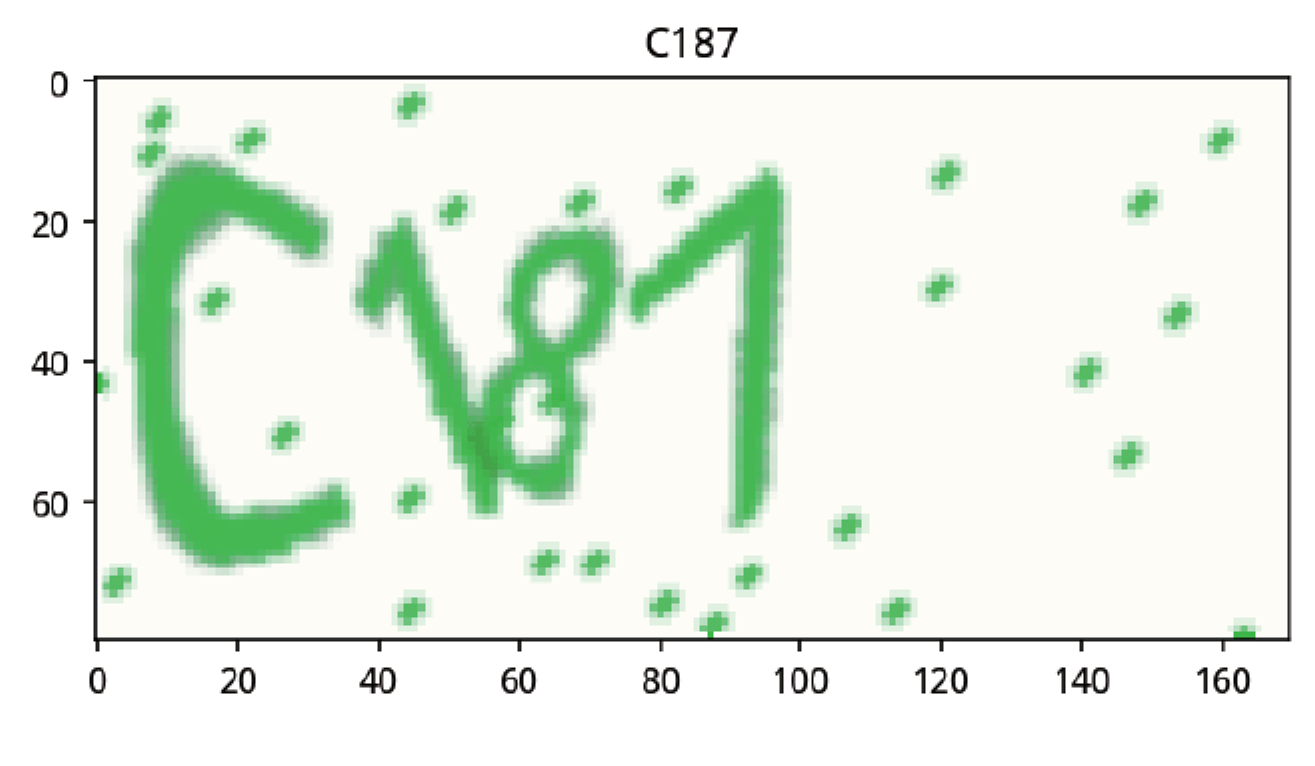


图 10-7 生成器输出的图像

这里：

- X 的 shape 是（1, 170, 80, 3），如果有 n 张图，shape 就是（n, 170, 80, 3）。
- y 是 label，可以看到生成的图片是 T4LL，那么按照上面的 characters，label 就是 [29 4 21 21]，外面还有一个框是因为这里可以有 n 个 label。
- input_length 表示模型输出的长度，这里是 15。
- label_length 表示 labels 的长度，这里是 4。

10.2.5 评估模型

接下来，通过这个函数来评估我们的模型，和上面的评估标准一样，只有全部正确，才算预测正确。这里有个坑，就是模型最开始训练的时候，并不一定会输出 4 个字符，所以如果遇到所有的字符都不到 4 个的时候，就不用计算了，一定是全错。遇到多于 4 个字符的时候，只取前 4 个。

```
def evaluate(batch_size=128, steps=10):
    batch_acc = 0
    generator = gen(batch_size)
    for i in range(steps):
        [X_test, y_test, _, _], _  = next(generator)
        y_pred = base_model.predict(X_test)
        shape = y_pred[:,2:,:].shape
        ctc_decode = K.ctc_decode(y_pred[:,2:,:],
                input_length=np.ones(shape[0])*shape[1]
        )[0][0]
        out = K.get_value(ctc_decode)[:, :n_len]
        if out.shape[1] == n_len:
            batch_acc += (y_test == out).all(axis=1).mean()
    return batch_acc / steps
```

10.2.6 评估回调

因为 Keras 没有针对 CTC 模型计算准确率的选项，因此需要自定义一个回调函数，它会在每一代训练完成的时候计算模型的准确率。

```
from keras.callbacks import *

class Evaluator(Callback):
    def __init__(self):
        self.accs = []

    def on_epoch_end(self, epoch, logs=None):
        acc = evaluate(steps=20)*100
        self.accs.append(acc)
        print('')
        print('acc: %f%%' % acc)

evaluator = Evaluator()
```

10.2.7 训练模型

先按照 Adam(1e-3) 的学习率训练 20 代，让模型快速收敛，然后以 Adam(1e-4) 的学习率再训练 20 代。这里设置每代训练 400 个 step，也就是每代 400*128=51200 个样本，验证集设置的是 20*128=2048 个样本。

```
h = model.fit_generator(gen(128),
                        steps_per_epoch=400,
                        epochs=20,
                        callbacks=[evaluator],
                        validation_data=gen(128),
                         validation_steps=20
)
model.compile(
   loss={'ctc': lambda y_true, y_pred: y_pred},
   optimizer=Adam(1e-4)
)

h2 = model.fit_generator(gen(128),
                        steps_per_epoch=400,
                        epochs=20,
                        callbacks=[evaluator],
                        validation_data=gen(128),
                        validation_steps=20
)
```

接下来，将 loss 和 acc 的曲线图画出来（见图 10-8）：

```
plt.figure(figsize=(10, 4))
plt.subplot(1, 2, 1)
plt.plot(h.history['loss'] + h2.history['loss'])
plt.plot(h.history['val_loss'] + h2.history['val_loss'])
plt.legend(['loss', 'val_loss'])
plt.ylabel('loss')
plt.xlabel('epoch')
plt.ylim(0, 1)

plt.subplot(1, 2, 2)
plt.plot(evaluator.accs)
plt.ylabel('acc')
plt.xlabel('epoch')
```

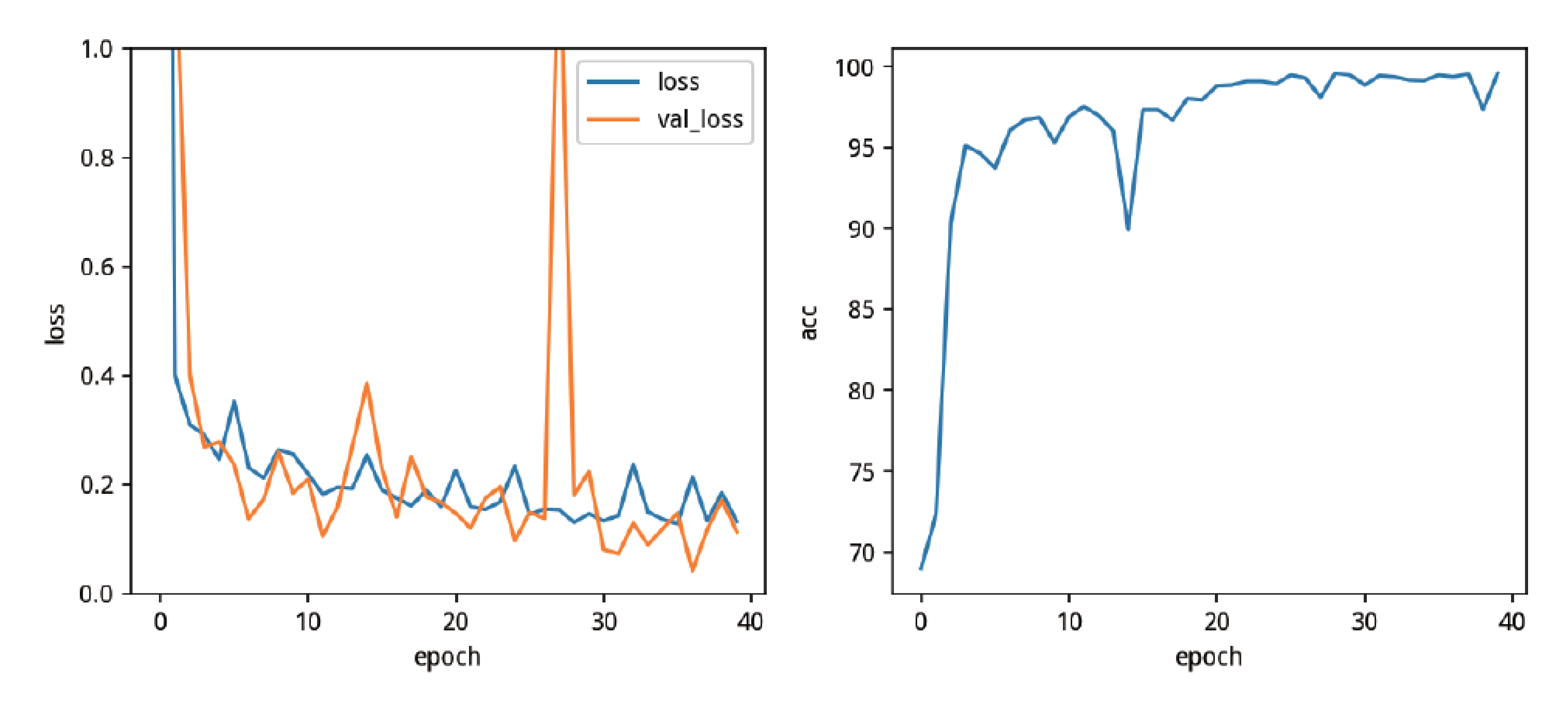


图 10-8 CNN+GRU+CTC 模型训练过程的 loss 和 acc 可视化

训练到 20 代的时候，模型表现如下：

```
Epoch 20/20
399/400 [============================>.] - ETA: 0s - loss: 0.1593
acc: 97.929688%
400/400 [==============================] - 122s - loss: 0.1589 - val_
loss: 0.1671
```

训练到 40 代的时候，模型表现如下：

```
Epoch 20/20
399/400 [============================>.] - ETA: 0s - loss: 0.1317
acc: 99.570312%
400/400 [==============================] - 123s - loss: 0.1315 - val_
loss: 0.1130
```

10.2.8 测试模型

测试模型的代码如下：

```
    (X_vis, y_vis, input_length_vis, label_length_vis), _ = next(gen(12))

    y_pred = base_model.predict(X_vis)
    shape = y_pred[:,2:,:].shape
    ctc_decode = K.ctc_decode(y_pred[:,2:,:], input_length=np.ones(shape[0])*
shape[1])[0][0]
    out = K.get_value(ctc_decode)[:, :4]

    plt.figure(figsize=(16, 8))
    for i in range(12):
        plt.subplot(3, 4, i+1)
        plt.imshow(X_vis[i].transpose(1, 0, 2))
    plt.title('pred:%s\nreal :%s' % (
              ''.join([characters[x] for x in out[i]]),
                  ''.join([characters[x] for x in y_vis[i]]))
       )
```

其结果如图 10-9 所示。

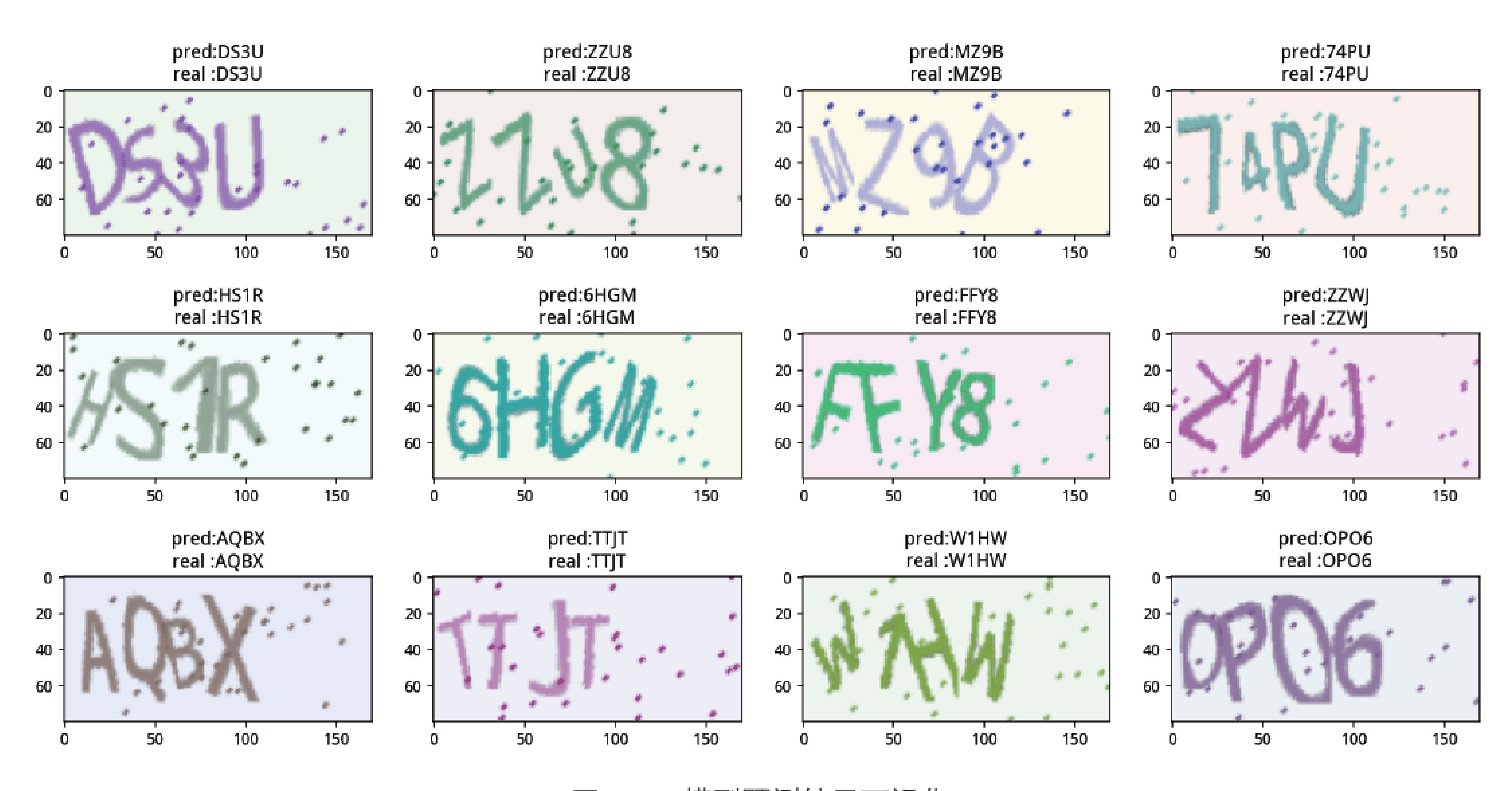


图 10-9 模型预测结果可视化

10.2.9 再次评估模型

我们可以尝试计算模型的总体准确率，以及查看模型到底错在哪儿。首先生成 1024 个样本，再用 base_model 进行预测，然后裁剪并进行 ctc 解码，最后裁剪到 4 个 label 并与真实值进行对比。

```
(X_vis, y_vis, input_length_vis, label_length_vis), _ = next(gen(10000))

y_pred = base_model.predict(X_vis, verbose=1)
shape = y_pred[:,2:,:].shape
ctc_decode = K.ctc_decode(y_pred[:,2:,:], input_length=np.
ones(shape[0])*shape[1])[0][0]
out = K.get_value(ctc_decode)[:, :4]

(y_vis == out).all(axis=1).mean()
```

运行结果：

```
# 0.99460000000000004
```

输出结果是 99.46% 的准确率，已经比上一个模型强很多了。

对预测错的样本进行统计：

```
from collections import Counter
Counter(''.join([characters[i] for i in y_vis[y_vis != out]]))
Counter({'0': 37, 'O': 14, 'Q': 1, 'T': 1, 'W': 1})
```

可以发现模型在 0 和 O 的准确率稍微低一点，其他的错误都只是个例。0 与 O 确实是很难分辨的，可以尝试用代码生成一个“'0O0O'”的图像，然后用模型进行预测：

```
characters2 = characters + ' '

generator = ImageCaptcha(width=width, height=height)
random_str = '0O0O'
X_test = np.array(generator.generate_image(random_str))
X_test = X_test.transpose(1, 0, 2)
X_test = np.expand_dims(X_test, 0)

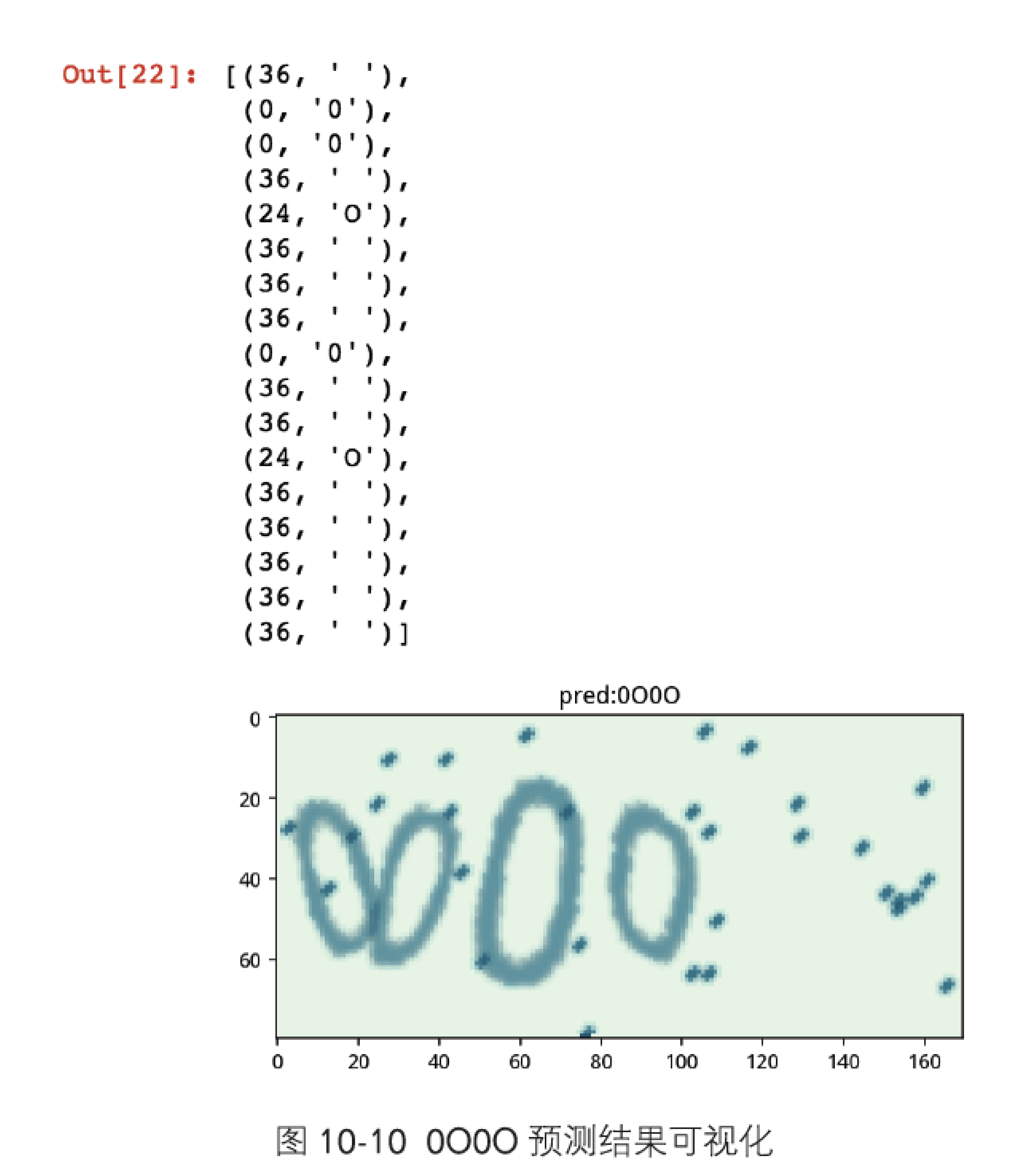
y_pred = base_model.predict(X_test)
shape = y_pred[:,2:,:].shape
ctc_decode = K.ctc_decode(y_pred[:,2:,:],
          input_length=np.ones(shape[0])*shape[1]
)[0][0]
out = K.get_value(ctc_decode)[:, :4]
out = ''.join([characters[x] for x in out[0]])

plt.imshow(X_test[0].transpose(1, 0, 2))
plt.title('pred:' + str(out))

argmax = np.argmax(y_pred, axis=2)[0]
list(zip(argmax, ''.join([characters2[x] for x in argmax])))
```

其结果如图 10-10 所示。

可以看到模型预测得还是很准的。



```
Out[22]: [(36, ' '),
          (0, '0'),
          (0, '0'),
          (36, ' '),
          (24, 'O'),
          (36, ' '),
          (36, ' '),
          (36, ' '),
          (0, '0'),
          (36, ' '),
          (36, ' '),
          (24, 'O'),
          (36, ' '),
          (36, ' '),
          (36, ' '),
          (36, ' '),
          (36, ' ')]
```


图 10-10　0O0O 预测结果可视化

10.2.10　总结

模型的大小是 3.3MB，在显卡上运行 10000 张验证码需要用 9 秒，平均一秒识别 1000 张以上，完全可以拼过网速。即使是在笔记本上运行，也可以运行到一秒几十张的速度，因此此类验证码可以说已经被破解了。

10.3　识别四则混合运算验证码（初赛）

在写完上面的内容之后不久，百度云和魅族联合办了一个深度学习的比赛，识别如图 10-11 所示的四则混合运算的表达式。（这个图片和之前做得很像，比赛结果是笔者被第一名在比赛最后一小时超了 3 个样本，拿到第二名。）



图 10-11　初赛样本之一

本节将详细介绍在进行四则混合运算识别竞赛初赛时的所有思路。核心思想在本章前两节内容中已经有所介绍，所以本节会省略部分重复内容。

10.3.1 问题描述

本次竞赛的目的是解决一个 OCR 问题，通俗地讲就是实现图像到文字的转换过程。

1. 数据集

初赛数据集一共包含 10 万张 180*60 的图片和一个 labels.txt 的文本文件。每张图片包含一个数学运算式，运算式包含：3 个运算数（3 个 0~9 的整型数字），2 个运算符（可以是 +、-、*，分别代表加法、减法、乘法），0 或 1 对括号（括号可能是 0 对或者 1 对）。

图片的名称从 0.png 到 99999.png，下面取出一张（见图 10-11）样例图片。文本文件 labels.txt 包含 10W 行文本，每行文本包含每张图片对应的公式以及公式的计算结果，公式和计算结果之间用空格分开，例如图 10-11 对应的文本如下所示：

5-6+2 1

2. 评价指标

官方的评价指标是准确率，初赛只有整数的加、减、乘运算，所得的结果一定是整数，所以要求序列与运算结果都正确才会判定为正确。

我们本地除了会使用官方的准确率作为评估标准以外，还会使用 CTC Loss 来评估模型。

10.3.2 数据集探索

根据题目要求，label 必定是三个数字、两个运算符、一对或没有括号，根据括号规则，只有可能是没括号、左括号和右括号，因此很容易就可以写出数据生成器的代码。

生成器的生成规则很简单：

```
import string
import random

digits = string.digits
operators = '+-*'
characters = digits + operators + '() '

def generate():
    seq = ''
    k = random.randint(0, 2)

    if k == 1:
        seq += '('
    seq += random.choice(digits)
    seq += random.choice(operators)
    if k == 2:
        seq += '('
```

```
        seq += random.choice(digits)
        if k == 1:
            seq += ')'
        seq += random.choice(operators)
        seq += random.choice(digits)
        if k == 2:
            seq += ')'

        return seq
```

相信大家都能看懂。当然，还有更好的写法，比如：

```
import random

def generate():
    ts = [u'{}{}{}{}{}', '({}{}{}){}{}', '{}{}({}{}{})']
    ds = u'0123456789'
    os = u'+-*'
    cs = [random.choice(ds) if x%2 == 0 else random.choice(os)
          for x in range(5)]
    return random.choice(ts).format(*cs)
```

除了生成算式以外，还有一个值得注意的地方就是初赛所有的减号（也就是'-'）都是细的，但是我们直接用 captcha 库生成图像会得到粗的减号，所以修改了 captcha python 库中 image.py 的代码，在 _draw_character 函数中增加了一句判断，如果是减号，就不进行 resize 操作，这样能够防止减号变粗：

```
# https://github.com/lepture/captcha/blob/v0.2.2/captcha/image.py
# line 191-194
if c != '-':
    im = im.resize((w2, h2))
    im = im.transform((w, h), Image.QUAD, data)

import string
import os

digits = string.digits
operators = '+-*'
characters = digits + operators + '() '
width, height, n_len, n_class = 180, 60, 7, len(characters) + 1
from captcha.image import ImageCaptcha
generator = ImageCaptcha(width=width, height=height,
    font_sizes=range(35, 56),
```

```
        fonts=['fonts/%s'%x for x in os.listdir('fonts') if '.tt' in x]
    )
    generator.generate_image('(1-2)-3')
```

图 10-12 就是原版生成器生成的图，可以看到减号是很粗的。我们再使用改写后的代码，生成如图 10-13 所示的图，可以看到减号已经不粗了。

```
    from image import ImageCaptcha
    generator = ImageCaptcha(width=width, height=height,
       font_sizes=range(35, 56),
    fonts=['fonts/%s'%x for x in os.listdir('fonts') if '.tt' in x]
    )
    generator.generate_image('(1-2)-3')
```



图 10-12 原版生成器生成的图



图 10-13 改进的生成器生成的图

10.3.3 模型结构

模型结构的代码如下：

```
    from keras.layers import *
    from keras.models import *
    from make_parallel import make_parallel
    rnn_size = 128

    input_tensor = Input((width, height, 3))
    x = input_tensor
    for i in range(3):
        x = Conv2D(32*2**i, (3, 3), kernel_initializer='he_normal')(x)
        x = BatchNormalization()(x)
        x = Activation('relu')(x)
        x = Conv2D(32*2**i, (3, 3), kernel_initializer='he_normal')(x)
        x = BatchNormalization()(x)
        x = Activation('relu')(x)
        x = MaxPooling2D(pool_size=(2, 2))(x)

    conv_shape = x.get_shape()
    x = Reshape(target_shape=(int(conv_shape[1]), int(conv_shape[2]*conv_
shape[3])))(x)

    x = Dense(128, kernel_initializer='he_normal')(x)
```

```
    x = BatchNormalization()(x)
    x = Activation('relu')(x)

    gru_1 = GRU(rnn_size, return_sequences=True,
             kernel_initializer='he_normal', name='gru1')(x)
    gru_1b = GRU(rnn_size, return_sequences=True, go_backwards=True,
             kernel_initializer='he_normal',
                    name='gru1_b')(x)
    gru1_merged = add([gru_1, gru_1b])

    gru_2 = GRU(rnn_size, return_sequences=True, kernel_initializer='he_
normal', name='gru2')(gru1_merged)
    gru_2b = GRU(rnn_size, return_sequences=True, go_backwards=True,
             kernel_initializer='he_normal',
                    name='gru2_b')(gru1_merged)
    x = concatenate([gru_2, gru_2b])
    x = Dropout(0.25)(x)
    x = Dense(n_class, kernel_initializer='he_normal', activation='softmax')(x)
    base_model = Model(input=input_tensor, output=x)

    base_model2 = make_parallel(base_model, 4)

    labels = Input(name='the_labels', shape=[n_len], dtype='float32')
    input_length = Input(name='input_length', shape=(1,), dtype='int64')
    label_length = Input(name='label_length', shape=(1,), dtype='int64')
    loss_out = Lambda(ctc_lambda_func, name='ctc')([base_model2.output,
labels, input_length, label_length])

    model = Model(inputs=(input_tensor, labels, input_length, label_length),
outputs=loss_out)
    model.compile(loss={'ctc': lambda y_true, y_pred: y_pred},
optimizer='adam')
```

模型结构像之前写的文章一样，只是把卷积核的个数改多了一点，并且做了一点小改动支持多 GPU 训练。如果你使用的是单卡，可以直接去掉 base_model2 = make_parallel(base_model, 4) 的代码：

```
    from keras.layers.merge import Concatenate
    from keras.layers.core import Lambda
    from keras.models import Model

    import tensorflow as tf

    def make_parallel(model, gpu_count):
        def get_slice(data, idx, parts):
            shape = tf.shape(data)
```

```
        size = tf.concat([ shape[:1] // parts, shape[1:] ],axis=0)
        stride = tf.concat([ shape[:1] // parts,
                             shape[1:]*0 ],
                             axis=0
    )
        start = stride * idx
        return tf.slice(data, start, size)

    outputs_all = []
    for i in range(len(model.outputs)):
        outputs_all.append([])

    # 每个 GPU 中复制一份模型， 然后共同分担同一批次（batch）输入数据
    for i in range(gpu_count):
        with tf.device('/gpu:%d' % i):
            with tf.name_scope('tower_%d' % i) as scope:
                inputs = []
                # 给各 GPU 平均分配任务，分析同一批次的输入数据
                for x in model.inputs:
                    input_shape = tuple(x.get_shape().as_list())[1:]
                    slice_n = Lambda(
                get_slice,
                output_shape=input_shape,
                arguments={'idx':i,'parts':gpu_count}
                )(x)
                    inputs.append(slice_n)

                outputs = model(inputs)
                if not isinstance(outputs, list):
                    outputs = [outputs]

                # 保存不同 GPU 的训练结果
                for l in range(len(outputs)):
                    outputs_all[l].append(outputs[l])

    # CPU 合并不同 GPU 的训练结果，返回模型
    with tf.device('/cpu:0'):
        merged = []
        for outputs in outputs_all:
            merged.append(Concatenate(axis=0)(outputs))

        return Model(model.inputs, merged)
```

base_model 的可视化如图 10-14 所示。

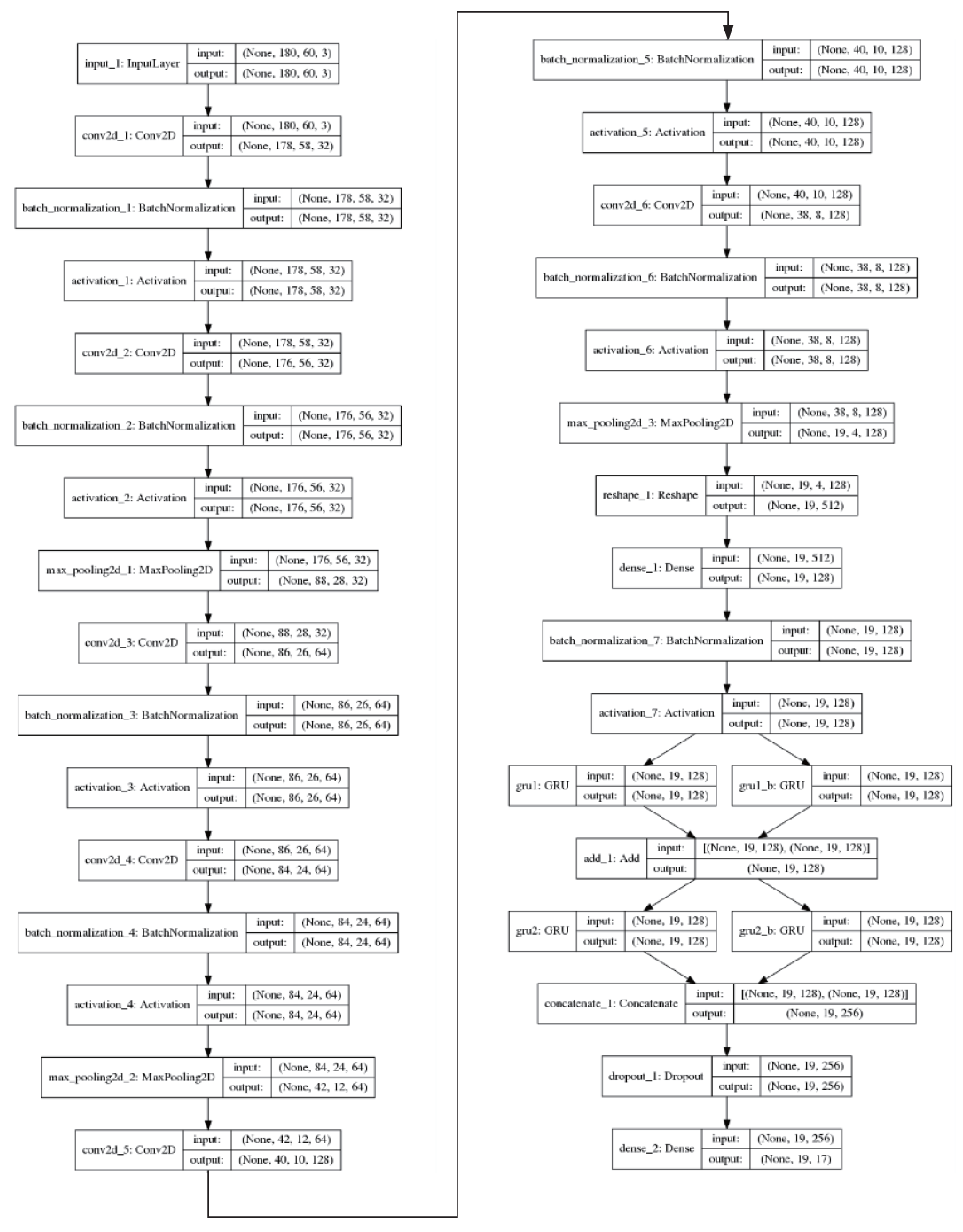


图 10-14 base_model 可视化

model 的可视化如图 10-15。

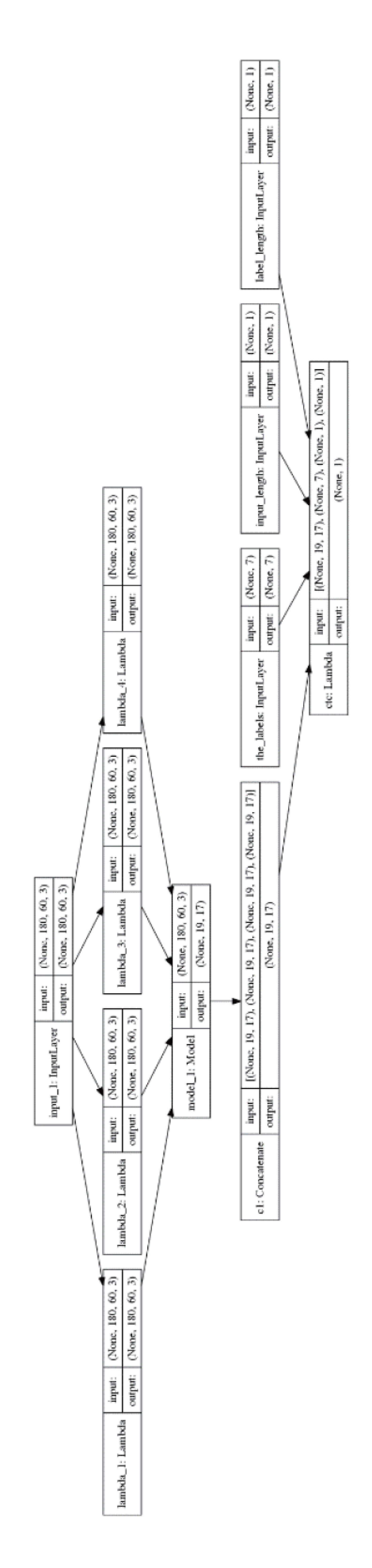


图 10-15 并行模型可视化

我们可以看到模型先分为 4 份，在 4 个显卡上并行计算，然后合并结果，计算最后的 CTC LOSS，进而训练模型。

在经过几次测试以后，抛弃 evaluate 函数，因为在验证集上已经能够做到 100% 识别率了，所以只需要看 val_loss 就可以了。在经过之前的几次尝试以后，发现在有生成器的情况下，训练代数越多越好，因此直接用 adam 运行了 50 代，每代 10 万样本，可以看到模型在 10 代以后基本已经收敛。虽然 30 代之前有一点振荡，但是到 30 代以后，模型已经稳定在 100% 识别率了，如图 10-16 所示。

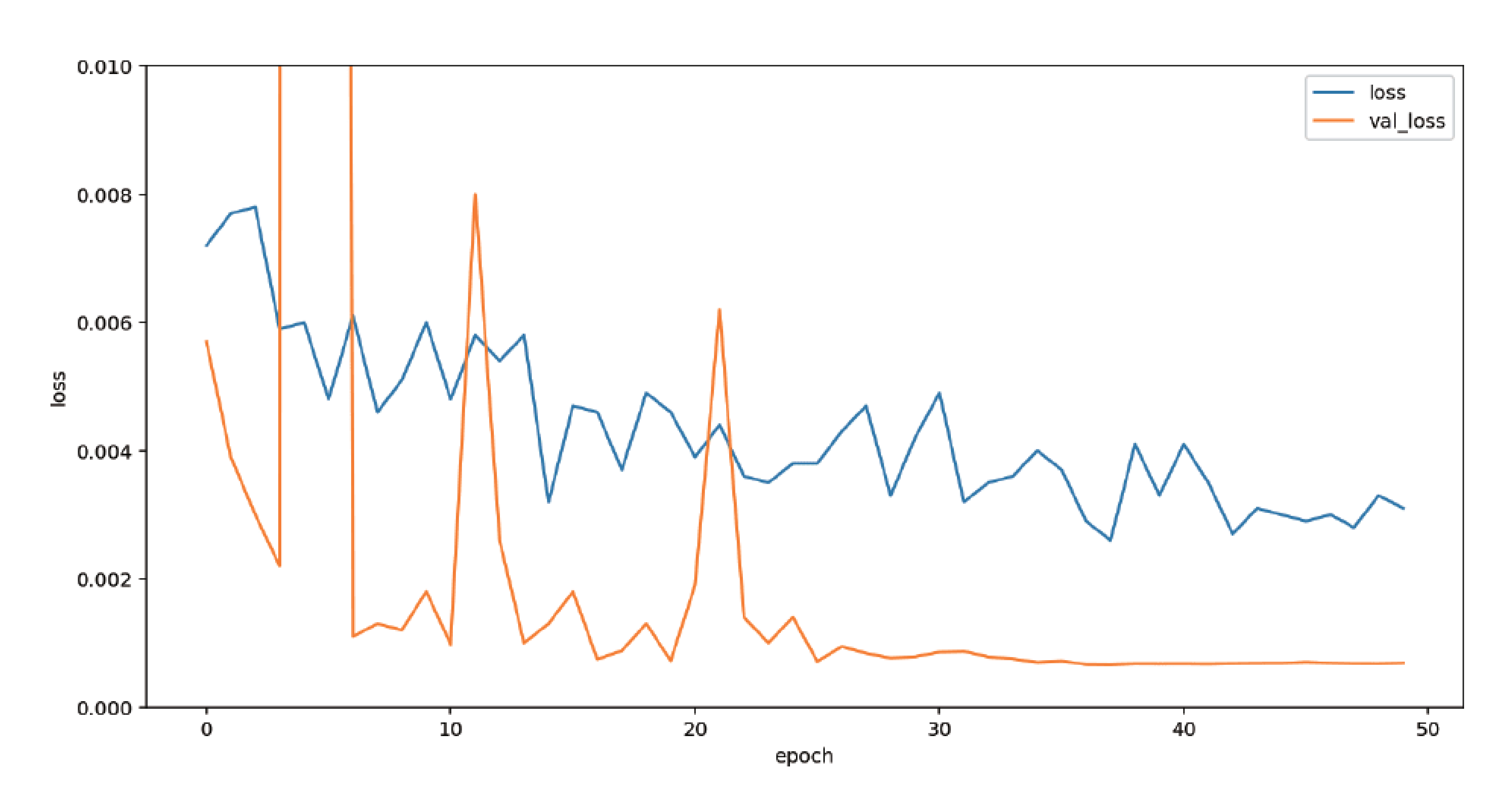


图 10-16　训练过程 loss 可视化

10.3.4　结果可视化

这里对生成的数据进行了可视化，可以看到模型基本已经做到万无一失，百发百中，如图 10-17 所示。

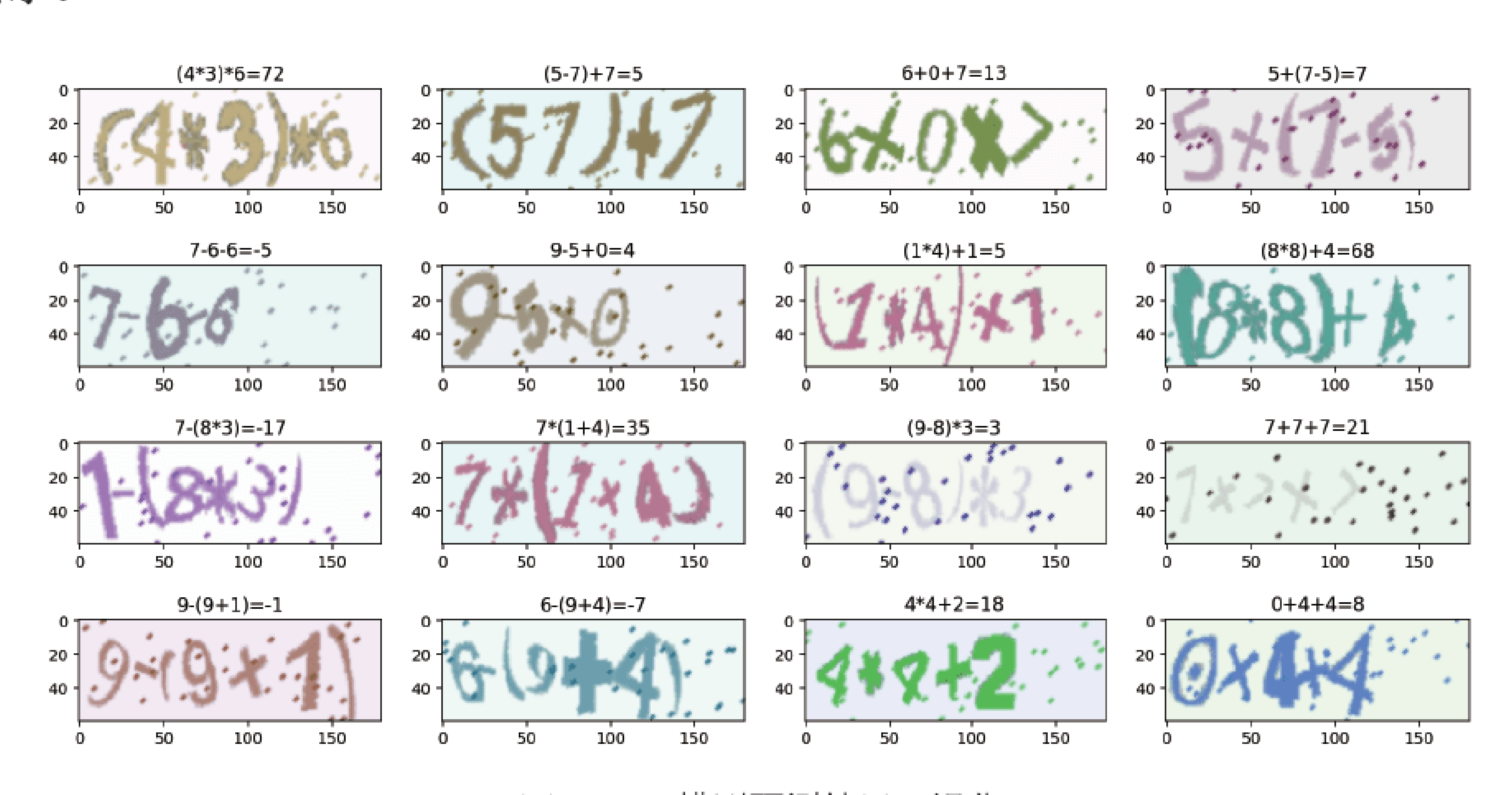


图 10-17　模型预测结果可视化

打包成 Docker 以后提交到比赛系统中，经过十几分钟的运行，我们得到了完美的 1 分，如图 10-18 所示。



图 10-18 提交结果截图

10.3.5 总结

初赛是非常简单的，因此才能得到这么准的分数，之后官方进一步提升了难度，将初赛测试集提高到 20 万张，在这个数据集上我们的模型只能拿到 0.999925 的成绩，可行的改进方法是将学习率进一步降低，充分训练模型，将多个模型结果融合等。

扩充测试集难点

在扩充数据集上，发现有一些图片预测出来无法计算，比如 [629,2271,6579,17416,71857,77631, 95303,102187,117422,142660,183693] 等，这里以 117422.png 为例（见图 10-19）。

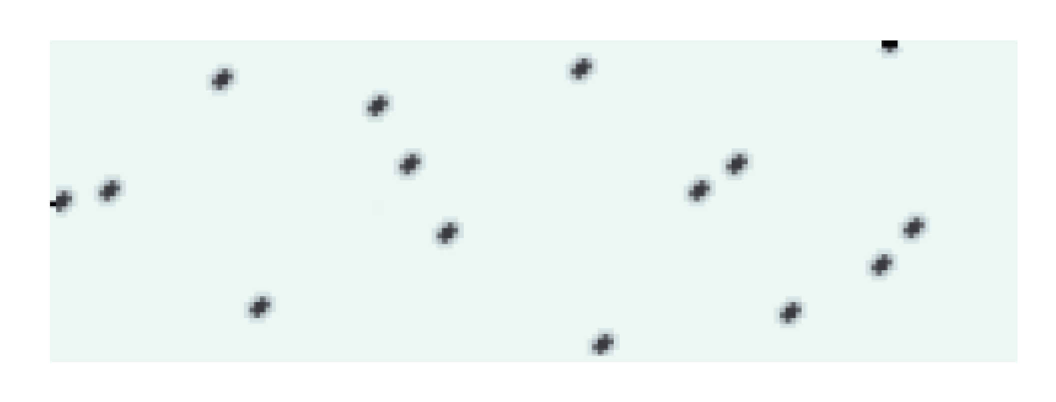


图 10-19 肉眼不可分辨但是机器可以分辨的样本

可以看到肉眼基本无法认出这个图，但是经过一定的图像处理后，能够显现出来它的真实面貌（见图 10-20）：

```
IMAGE_DIR = 'image_contest_level_1_validate'
index = 117422
img = cv2.imread('%s/%d.png' % (IMAGE_DIR, index))
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
h = cv2.equalizeHist(gray)
```

可以看到如图 10-20 所示的结果。

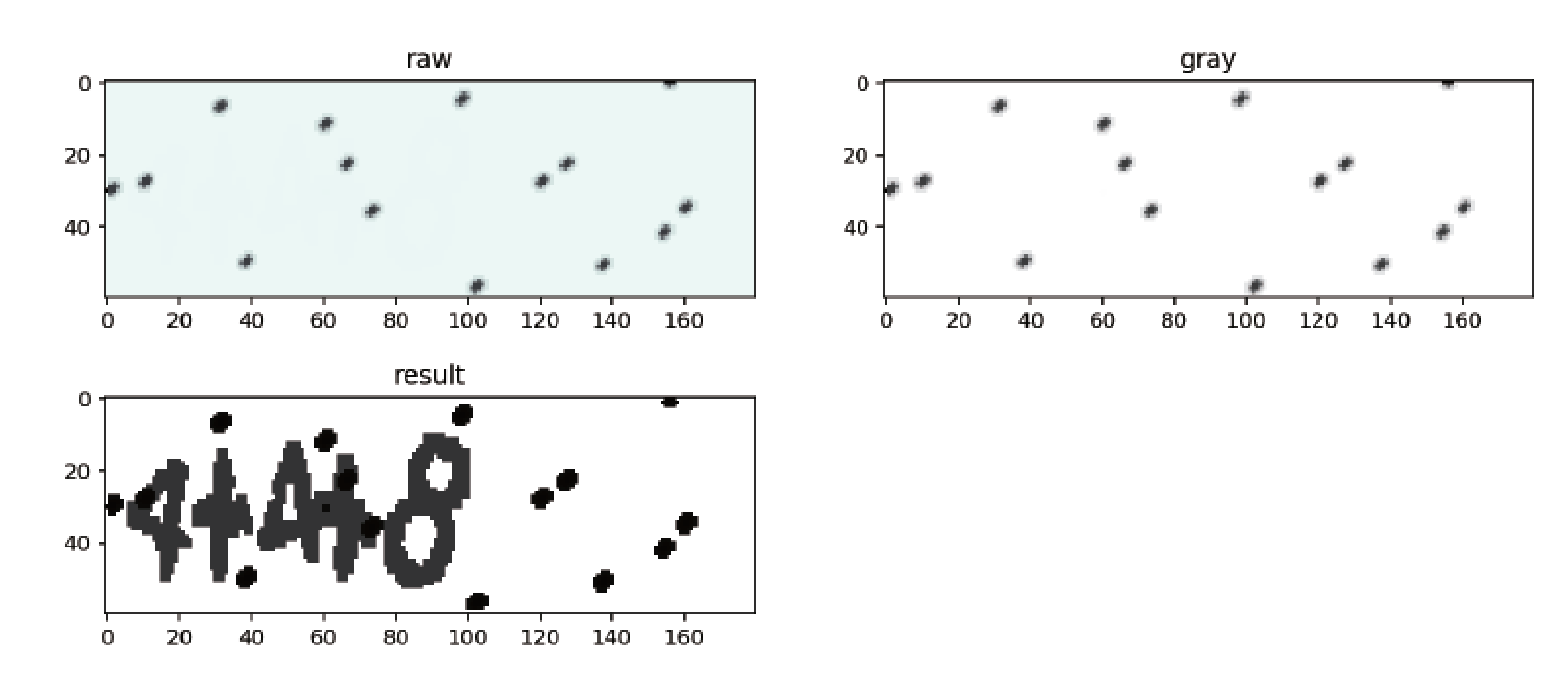


图 10-20　经过直方图均衡化以后的图像可视化

当然，还有一张图是无法通过预处理得到结果的，这是第 142660 个样本（见图 10-21）。



图 10-21　完全无法辨认的样本

这有可能是程序的 BUG 造成的小概率事件，所以初赛除了我们跑了一个 Docker 得到满分以外，没有第二个人达到满分。

10.4 识别四则混合运算验证码（决赛）

本节将详细介绍四则混合运算识别竞赛决赛时的所有思路。

10.4.1 问题描述

本次竞赛的目的是为了解决一个 OCR 问题，通俗地讲就是实现图像到文字的转换过程。

1. 数据集

决赛数据集一共包含 10 万张图片和一个 labels.txt 的文本文件。每张图片包含一个数学运算式，运算式中包含：

（1）图片大小不固定。

（2）图片中的某一块区域为公式部分。

（3）图片中包含二行或者三行的公式。

（4）公式类型有两种：赋值和四则运算的公式。两行的包括一个赋值公式和一个计算公式，三行的包括两个赋值公式和一个计算公式。加号（+）即使旋转为 x，仍为加号，* 是乘号。

（5）赋值类的公式，变量名为一个汉字。 汉字来自两句诗（不包括逗号）： 君不见，黄河之水天上来，奔流到海不复回 烟锁池塘柳，深圳铁板烧。

（6）四则运算的公式包括加法、减法、乘法、分数、括号。 其中的数字为多位数字，汉字为变量，由上面的语句赋值。

（7）输出结果的格式为：图片中的公式，一个英文空格，计算结果。其中： 不同行公式之间使用英文分号分隔计算结果时，分数按照浮点数计算，计算结果误差不超过 0.01，视为正确。

（8）整个 label 文件使用 UTF8 编码。

决赛样例如图 10-22 所示。

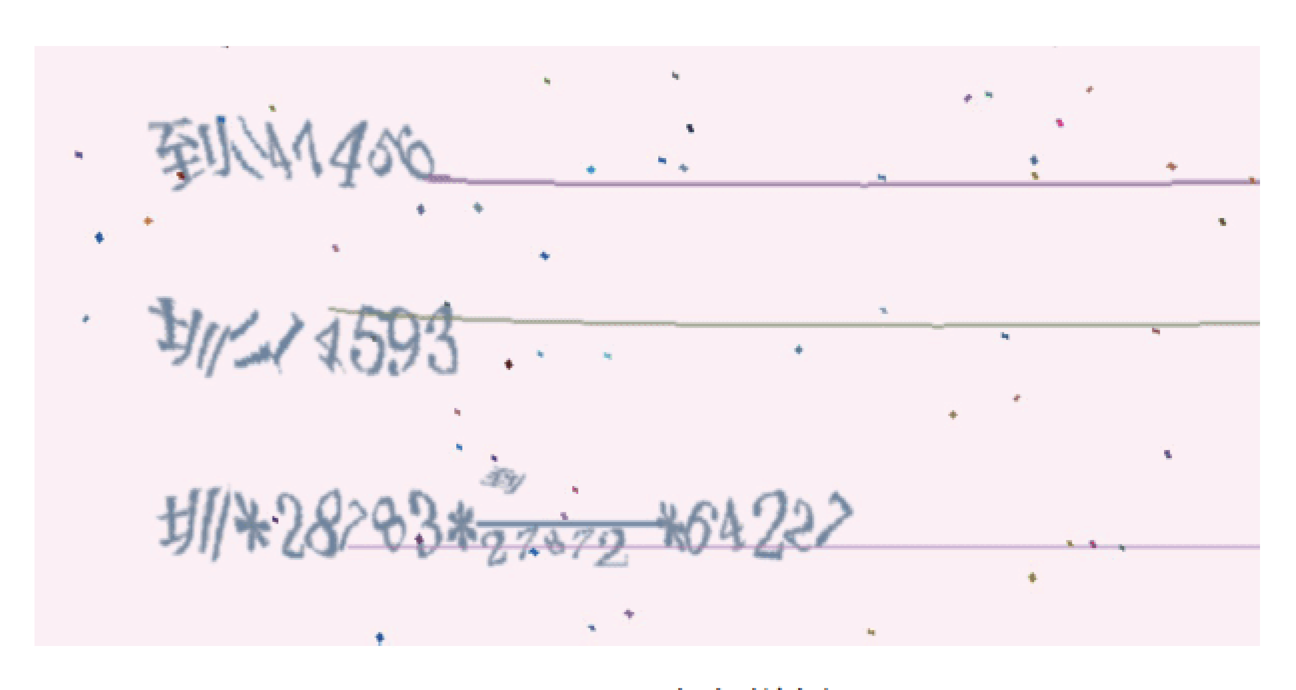


图 10-22 决赛样例

初赛的题不难，只需要识别文本序列即可，决赛的算式比较复杂，需要先经过图像处理，然后才能输入到神经网络中进行端到端的文本序列识别。

2. 评价指标

官方的评价指标是准确率，初赛只有整数的加减乘运算，所得的结果一定是整数，所以要求序列与运算结果都正确才会判定为正确。

决赛的数字通常都是 5 位数，并且会有很多乘法和加法，以及一定会存在的一个分数，所以结果很容易超出 64 位浮点数所能表示的范围，因此官方在经过讨论后决定只考虑文本序列的识别，不评价运算结果。

我们本地除了会使用官方的准确率作为评估标准以外，还会使用 CTC Loss 来评估模型。

10.4.2 数据集探索

1. 定义

决赛的数据集探索复杂得多，先明确两个概念：

流 =42072; 圳 =86;(圳 -(97510*45921))* 流 /35864

在这个式子中，“流 =42072; 圳 =86;”被称为赋值式，“(圳 -(97510*45921))* 流 /35864”被称为表达式，赋值式和表达式统称为公式，“+、-、*、/”被称为运算符。

2. 分析

首先对样本中每个字出现的次数进行了统计，如图 10-23 所示。

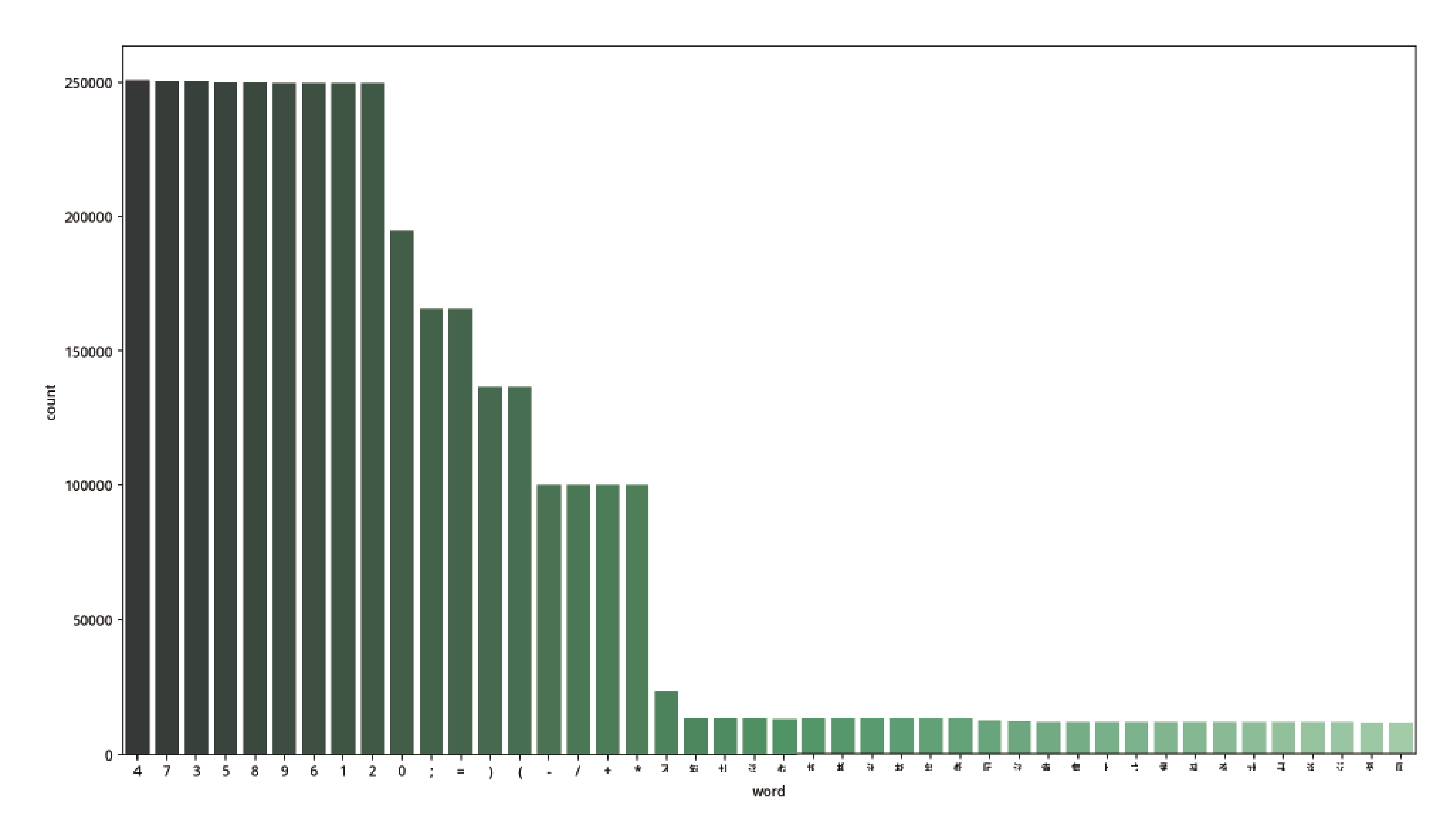


图 10-23　对字数进行统计

可以看到数字的分布很有意思，0 出现的次数比其他数字都低，其余的数字出现次数基本一样，立即推算这是直接按随机数生成的，0 不能出现在首位，所以概率变低。

分号和等号出现的次数一样，这是因为每个赋值式都有一个等号和一个分号。它出现的概率是 1.65807，因此可以猜出一个赋值式和两个赋值式的比例是 1:2。

运算符出现的概率都是一样的，所以可以推断它们是直接随机取的。

括号出现的概率是 1.36505，统计了一下括号出现的所有可能：

```
1+1+1+1

(1+1)+1+1
1+(1+1)+1
1+1+(1+1)
(1+1+1)+1
1+(1+1+1)

((1+1)+1)+1
(1+(1+1))+1

1+((1+1)+1)
1+(1+(1+1))

(1+1)+(1+1)
```

一共有 11 种可能，按照括号的数量统计括号出现的频率可以得出 2*5/11.0+5/11.0=1.3636，因此括号也是从上面几种模板随机取的。

中文除了“不”字出现了两次，概率翻倍，其他字概率基本相等。中文字取自于下面两句诗：“君不见，黄河之水天上来，奔流到海不复回 烟锁池塘柳，深圳铁板烧”，所以也可以推断出是按照字直接随机取的。

3. 小结

- 中文直接等概率取，“不”概率加倍。
- 括号从 11 种情况中随机取。
- 运算符每次必出 4 个。
- 1/3 概率取一个赋值式，2/3 概率取 2 个赋值式。
- 运算符 / 永远都会出现一次，中文在上。
- 运算符 +-* 随机取，概率都是 1/3。
- 数字取值范围是 [0, 100000]。

10.4.3 数据预处理

由于原始的图像十分巨大，直接输入到 CNN 中会有 90% 以上的区域是没有用的，因此需要对图像进行预处理，裁剪出有用的部分。接下来，因为图像有两到三个式子，因此采取的方案是从左到右拼接在一起，这样的好处是图像比较小（900*80=72000 vs 600*270=162000）。

这里主要使用了以下几种技术，这些技术的主要内容在第 3 章图像处理入门中基本都有所涉及：

- 转灰度图
- 直方图均衡
- 中值滤波
- 开闭运算
- 二值化
- 轮廓查找
- 边界矩形

参考链接：

http://docs.opencv.org/master/df/d9d/tutorial_py_colorspaces.html
http://docs.opencv.org/master/d5/daf/tutorial_py_histogram_equalization.html
http://docs.opencv.org/master/d4/d13/tutorial_py_filtering.html
http://docs.opencv.org/master/d9/d61/tutorial_py_morphological_ops.html
http://docs.opencv.org/master/d7/d4d/tutorial_py_thresholding.html
http://docs.opencv.org/master/d4/d73/tutorial_py_contours_begin.html
http://docs.opencv.org/master/dd/d49/tutorial_py_contour_features.html

首先进行初步的关键区域提取，操作步骤如下：

```
def plot(index):
    img = cv2.imread('%s/%d.png'%(IMAGE_DIR, index))
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    eq = cv2.equalizeHist(gray)
    b = cv2.medianBlur(eq, 9)

    m, n = img.shape[:2]
    b2 = cv2.resize(b, (n//4, m//4))

    m1 = cv2.morphologyEx(b2, cv2.MORPH_OPEN, np.ones((7, 40)))
    m2 = cv2.morphologyEx(m1, cv2.MORPH_CLOSE, np.ones((4, 4)))
    _, bw = cv2.threshold(m2, 127, 255, cv2.THRESH_BINARY_INV)

    bw = cv2.resize(bw, (n, m))

    r = img.copy()
    img2, ctrs, hier = cv2.findContours(bw, cv2.RETR_EXTERNAL,
      cv2.CHAIN_APPROX_SIMPLE)
    for ctr in ctrs:
        x, y, w, h = cv2.boundingRect(ctr)
        cv2.rectangle(r, (x, y), (x+w, y+h), (0, 255, 0), 10)
```

1. 去噪

拿到原始图像（图 10-24 左上角的图）后，首先将图像转灰度图，然后使用初赛的直方图均衡提高图像的对比度，结果在图 10-24 正中间的 eq。这里噪点还在，所以需要进行滤波，这里使用了中值滤波，它能够很好地滤掉噪点和干扰线，滤波结果在图 10-24 右上角 blur。

2. 连接公式

现在只关心公式的提取，而不在意字符的提取（因为无法保证准确提取），所以需要将这些字符连接起来。这里首先对图像进行 4 倍的缩放，结果是图 10-24 左下角的 m1。然后使用一种叫作开闭运算的算法来连接字符。因为要的是横向连接，纵向不需要连接，所以选择（7, 40）大小的开运算，为了滤掉不必要的噪声，我们使用（4, 4）的闭运算，开闭运算的结果位于图 10-24 中间的 m2。

3. 关键区域提取

在拼接好公式以后，就可以对图像使用轮廓查找的算法了，很容易抓到图像的三个边缘点集，然后使用边界矩形函数得到矩形的（x, y, w, h），完成关键区域提取。提取之后将绿色的矩形画在原图上，结果位于图 10-24 右下角的 rect。

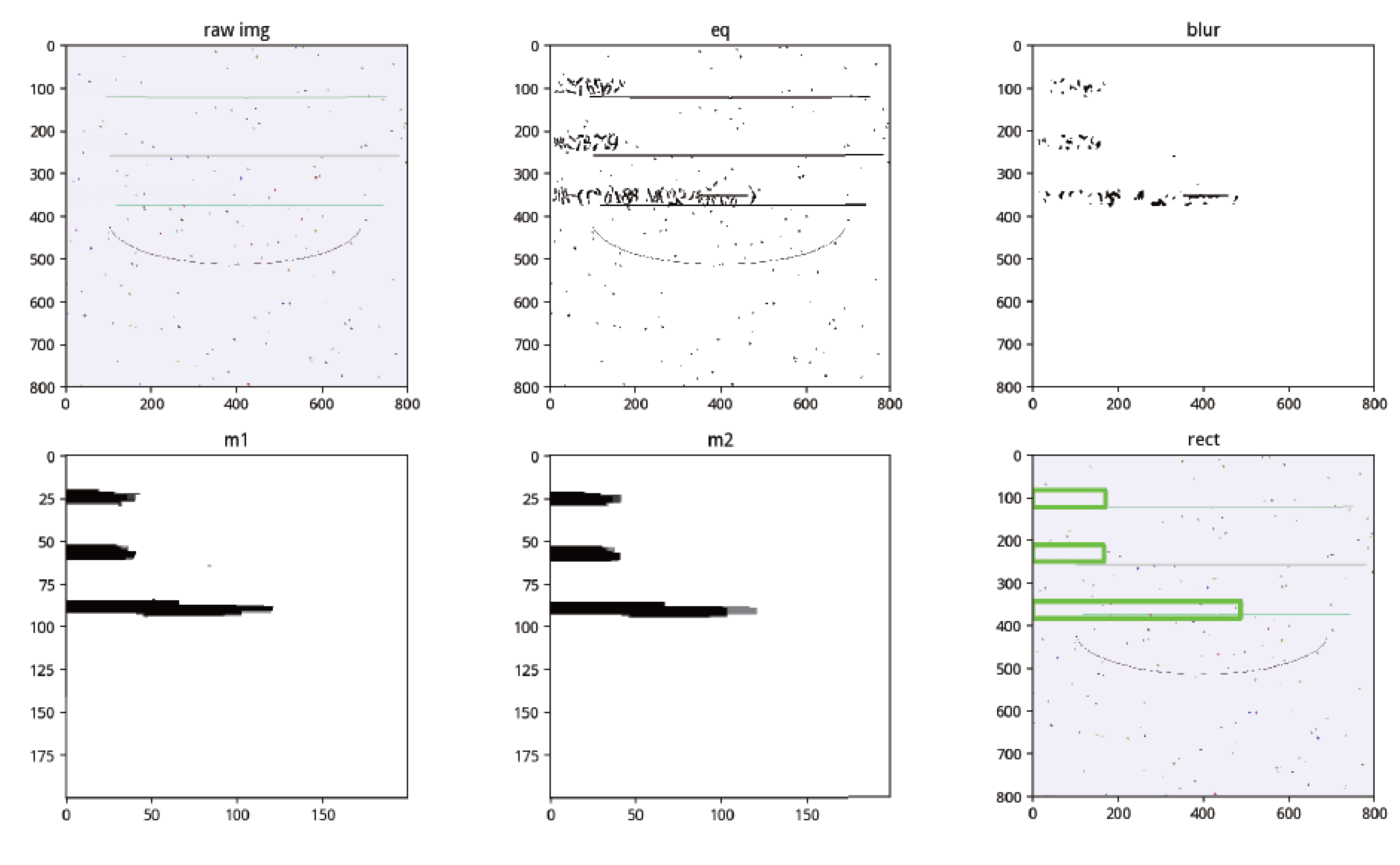


图 10-24 实施图像处理的过程

4. 微调

由于之前使用了很大的 kernel 进行滤波，会造成原始图像分辨率降低，因此这里需要进行一个微调的操作：

```
# 微调三个公式
d = 20
d2 = 5
imgs = []
sizes = []
for i, ctr in enumerate(ctrs):
    x, y, w, h = cv2.boundingRect(ctr)
    roi = img[max(0, y-d):min(m, y+h+d),max(0, x-d):min(n, x+w+d)]
    p, q, _ = roi.shape

    x = b[max(0, y-d):min(m, y+h+d),max(0, x-d):min(n, x+w+d)]
    x = cv2.morphologyEx(x, cv2.MORPH_CLOSE, np.ones((3, 3)))
    _, x = cv2.threshold(x, 127, 255, cv2.THRESH_BINARY_INV)
_, x, _ = cv2.findContours(x, cv2.RETR_EXTERNAL,
                cv2.CHAIN_APPROX_SIMPLE
)
    x, y, w, h = cv2.boundingRect(np.vstack(x))
roi2 = roi[max(0, y-d2):min(p, y+h+d2),
         max(0, x-d2):min(q, x+w+d2)]
    imgs.append(roi2)
    sizes.append(roi2.shape)
```

首先通过之前的矩形，扩充 20 像素，然后裁剪出关键区域，这里是直接对滤波的图裁剪，所以分辨率很高。接下来，经过简单的闭运算滤波，二值化，提取边框，这里即使有噪点也不用担心，裁多了不要紧，裁少了才麻烦，裁出来的图可能会比较小，因为滤波过了，所以再扩充 5 个像素，从而达到不错的效果。

图 10-25 中显示的是几个例子。左边是检测边框，中间是直接按照框切割，右边是扩充像素切割。

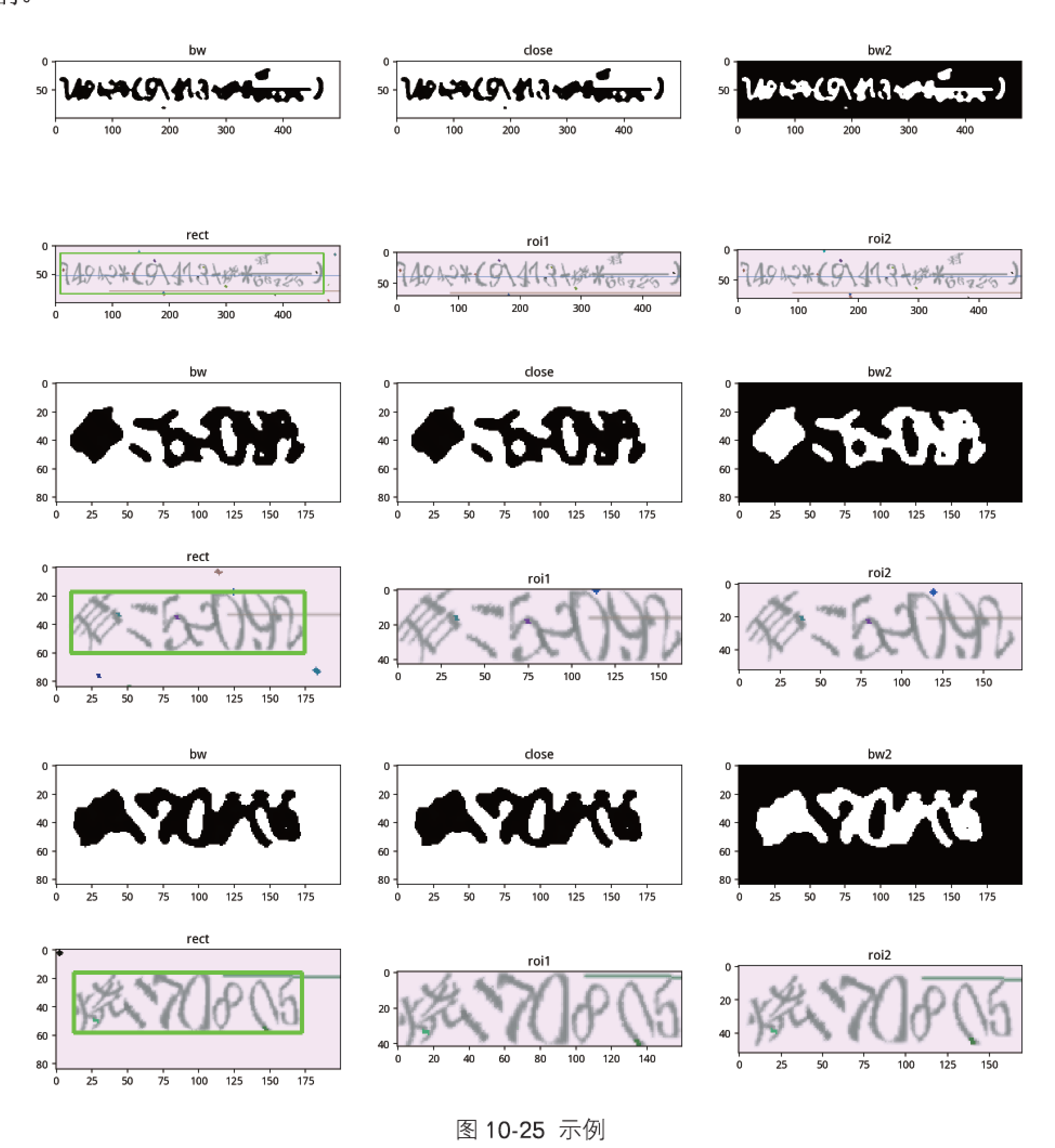


图 10-25　示例

5. 连接三个公式

裁出准确的公式后，就可以直接进行横向连接了：

```
# 连接三个公式
sizes = np.array(sizes)
img2 = np.zeros(
```

```
            (sizes[:,0].max(),
            sizes[:,1].sum()+2*(len(sizes)-1),
            3), dtype=np.uint8
)
x = 0
for a in imgs[::-1]:
    w = a.shape[1]
    img2[:a.shape[0], x:x+w] = a
    x += w + 2
```

如图 10-26 所示是拼接好的图像。

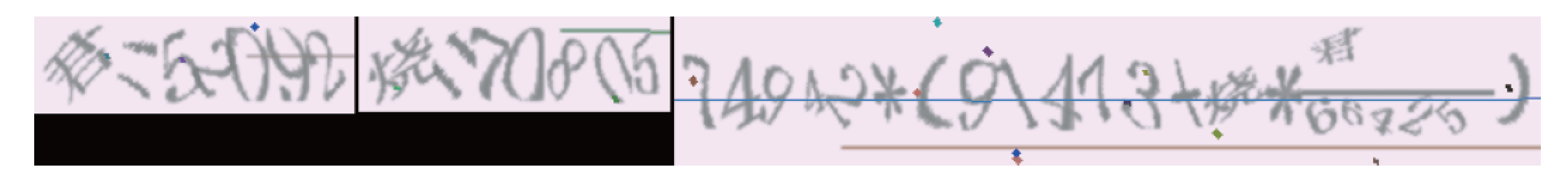


图 10-26 拼接好的图像

6. 并行预处理（见图 10-27）

如果直接使用 Python 的 for 循环去运行，只能占用一个核的 CPU 利用率，为了充分利用 CPU，使用多进程并行预处理的方法让每个 CPU 都能满载运行。为了能够实时查看进度，使用 tqdm 这个进度条的库。

```
p = Pool(12)

n = 100000
if __name__ == '__main__':
    rs = []
    for r in tqdm(p.imap_unordered(f, range(n)), total=n):
        rs.append(r)
```

```
In [3]: %%time

        try:
            p
        except:
            p = Pool(12)

        n = 100000
        if __name__ == '__main__':
            rs = []
            for r in tqdm(p.imap_unordered(f, range(n)), total=n):
                rs.append(r)

        100%|██████████| 100000/100000 [18:40<00:00, 89.28it/s]

        CPU times: user 17.1 s, sys: 2.79 s, total: 19.9 s
        Wall time: 18min 40s
```

图 10-27 并行预处理数据

7. 小结

这里将各个量之间的关系都画出来了（见图 10-28），很有意思。

```
pd.plotting.scatter_matrix(df, alpha=0.1, figsize=(14,8), diagonal='kde');
```

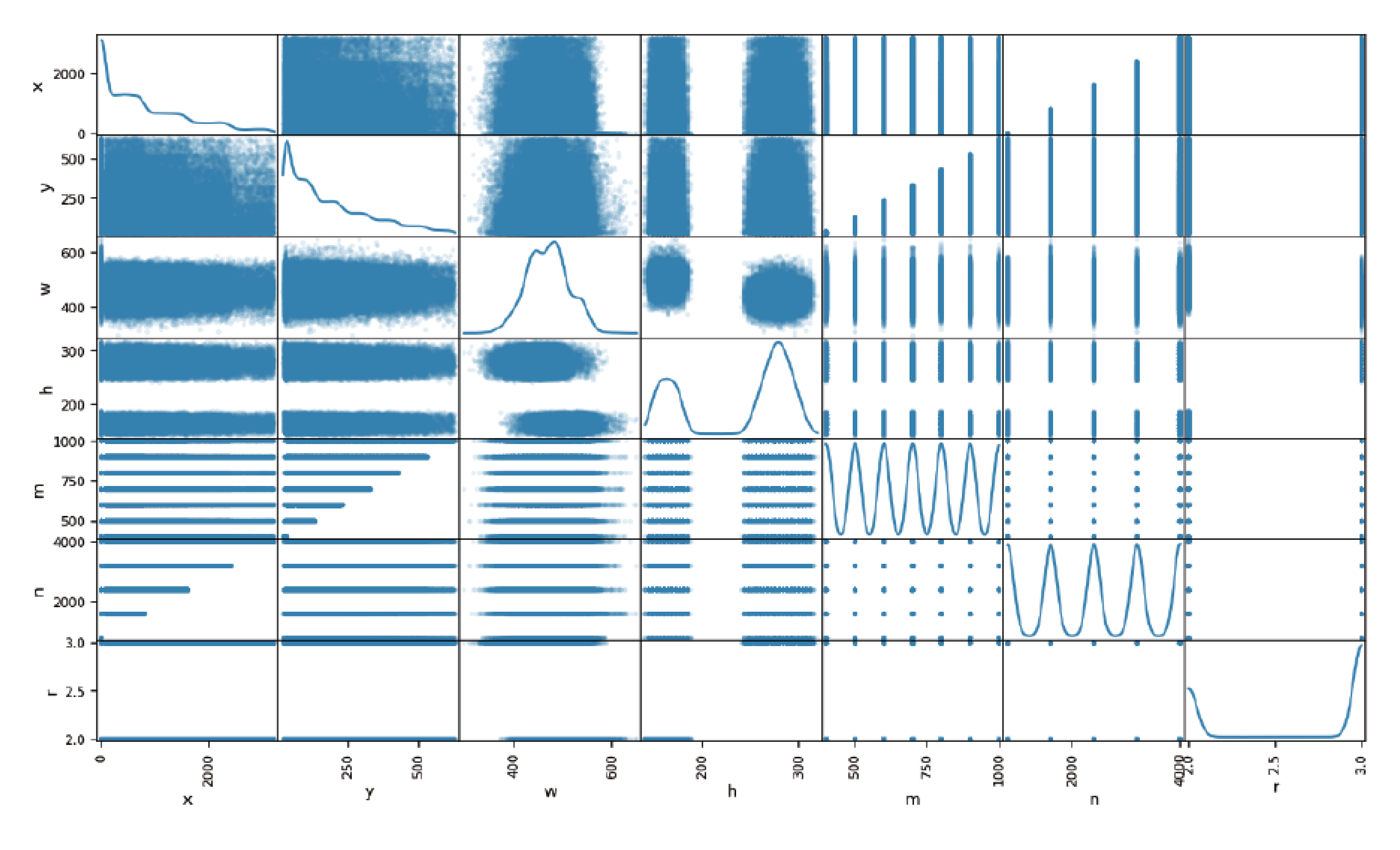


图 10-28　散布矩阵可视化

其中的 x, y 表示公式的起始坐标，w, h 表示公式的宽和高，n, m 表示原图的宽和高，r 表示有几个公式。我们可以从中看到，x, y 没有明显的规律，稍微有一点规律就是越宽的图能得到的 x 越大（宽 1000 的图不可能有公式出现在 1200）。

w 也没有明显的规律，是典型的正态分布，而 h 则有两个峰，这是因为公式有两个和三个的差别。

m, n 很有规律，它们是按照某几个固定的数随机取的，m 的取值是从 [400, 500, 600, 700, 800, 900, 1000] 中随机选取的，n 是从 [800, 1600, 2400, 3200, 4000] 中随机取的。

```
    Counter(df['m'])
    Counter({400: 14233,
             500: 14414,
             600: 14332,
             700: 14304,
             800: 14293,
             900: 14299,
             1000: 14125})


    Counter(df['n'])
    Counter({800: 19872, 1600: 19937, 2400: 20128, 3200: 19975, 4000:
20088})
```

10.4.4 模型结构

由于仅对 base_model 进行了修改，ctc 部分直接照搬之前的代码即可，因此这里只讨论 base_model，下面是代码：

```
    def ctc_lambda_func(args):
        y_pred, labels, input_length, label_length = args
        y_pred = y_pred[:, 2:, :]
        return K.ctc_batch_cost(labels, y_pred, input_length, label_length)

    rnn_size = 128
    l2_rate = 1e-5

    input_tensor = Input((width, height, 3))
    x = input_tensor
    for i, n_cnn in enumerate([3, 4, 6]):
        for j in range(n_cnn):
            x = Conv2D(32*2**i, (3, 3), padding='same', kernel_
initializer='he_uniform',
                       kernel_regularizer=l2(l2_rate))(x)
            x = BatchNormalization(gamma_regularizer=l2(l2_rate), beta_
regularizer=l2(l2_rate))(x)
            x = Activation('relu')(x)
        x = MaxPooling2D((2, 2))(x)

    # x = AveragePooling2D((1, 2))(x)
    cnn_model = Model(input_tensor, x, name='cnn')

    input_tensor = Input((width, height, 3))
    x = cnn_model(input_tensor)

    conv_shape = x.get_shape().as_list()
    rnn_length = conv_shape[1]
    rnn_dimen = conv_shape[3]*conv_shape[2]

    print conv_shape, rnn_length, rnn_dimen

    x = Reshape(target_shape=(rnn_length, rnn_dimen))(x)
    rnn_length -= 2
    rnn_imp = 0

    x = Dense(rnn_size, kernel_initializer='he_uniform', kernel_
regularizer=l2(l2_rate), bias_regularizer=l2(l2_rate))(x)
```

```
    x = BatchNormalization(gamma_regularizer=l2(l2_rate), beta_
regularizer=l2(l2_rate))(x)
    x = Activation('relu')(x)
    # x = Dropout(0.2)(x)

    gru_1 = GRU(rnn_size, implementation=rnn_imp, return_sequences=True,
name='gru1')(x)
    gru_1b = GRU(rnn_size, implementation=rnn_imp, return_sequences=True,
go_backwards=True, name='gru1_b')(x)
    gru1_merged = add([gru_1, gru_1b])

    gru_2 = GRU(rnn_size, implementation=rnn_imp, return_sequences=True,
name='gru2')(gru1_merged)
    gru_2b = GRU(rnn_size, implementation=rnn_imp, return_sequences=True,
go_backwards=True, name='gru2_b')(gru1_merged)
    x = concatenate([gru_2, gru_2b])

    # x = Dropout(0.2)(x)
    x = Dense(n_class, activation='softmax', kernel_regularizer=l2(l2_rate),
bias_regularizer=l2(l2_rate))(x)
    rnn_out = x
    base_model = Model(input_tensor, x)
```

在经过多次的代码迭代以后，将 cnn 打包为一个 model，这样模型会简洁很多。

模型思路是这样的：首先输入一张图，然后通过 cnn 导出（112, 10, 128）的特征图。其中，112 就是输入到 rnn 的序列长度，10 指的是每一条特征的高度为 10 像素，将后面（10, 128）的特征合并成 1280，然后经过一个全连接降维到 128 维，就得到了（112, 128）的特征，输入到 RNN 中，经过两层双向 GRU 输出 112 个字的概率，最后用 CTC LOSS 去优化模型，从而得到能够准确识别字符序列的模型，如图 10-29 所示。

1. CNN

在 CNN 中，理论上最大序列的长度为 46 个字符（数字可能为 100000），所以是 2*9+3*6+4+4+2=46。对于 CTC 来说，最好输入大于最大长度 2 倍的序列，才能收敛得比较好。之前直接卷积到 50 左右，然后对于连续字符来说，没有空白能将它们分隔开来，所以收敛效果会差很多。需要注意的是最大序列长度，这里用 Python2 之前总是算错，因为没有 Decode 成 UTF-8 的话，一个中文占三个字节。

CNN 的结构由原来的两层卷积一层池化，改为了多层卷积一层池化的结构。由于卷积层分别是 3、4 和 6 层，因此称之为 346 结构。

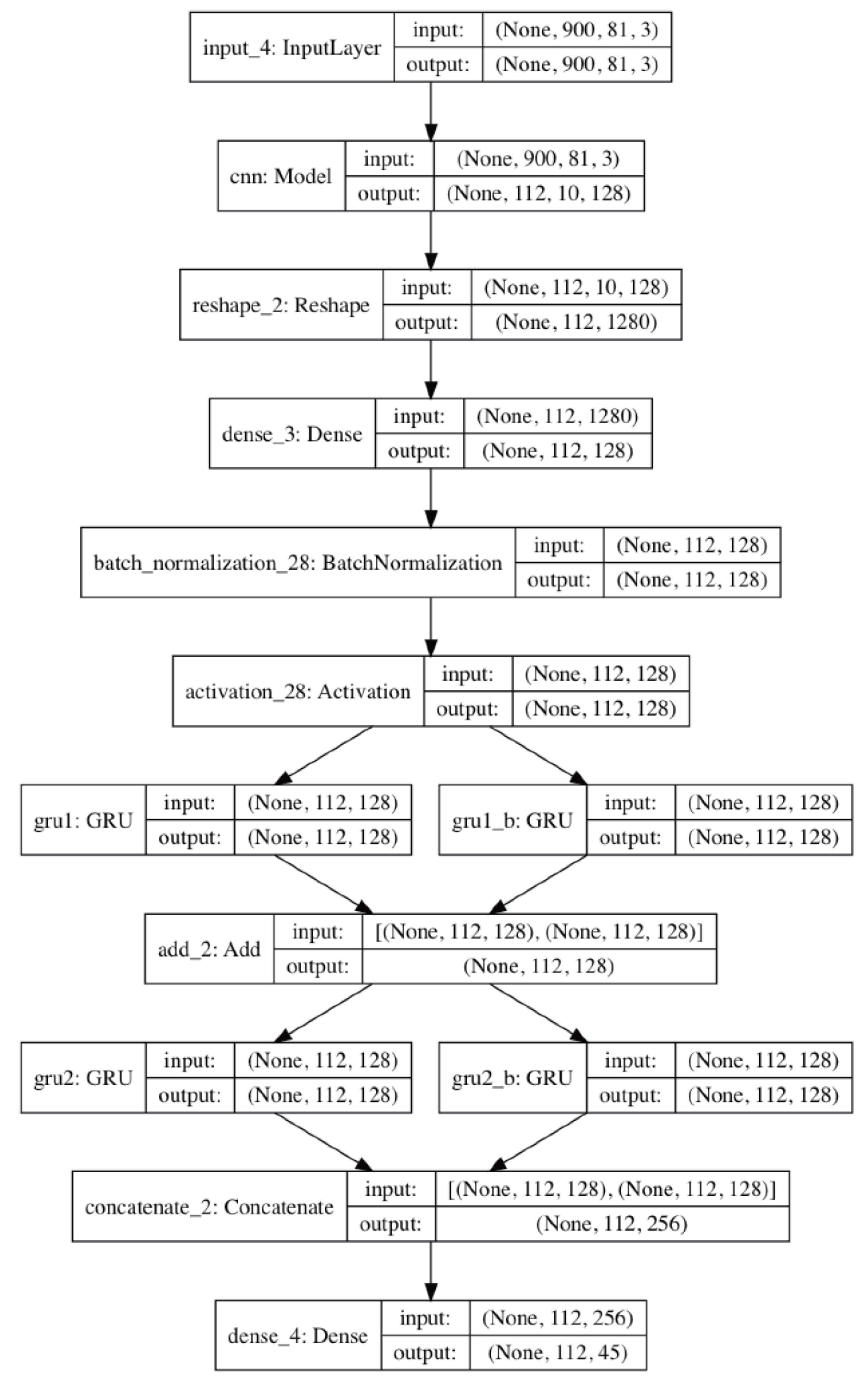


图 10-29 模型结构可视化

2. GRU

为什么使用循环神经网络呢？这里举一个经典的例子：“研表究明，汉字的序顺并不定一能影阅响读”，当看完这句话后，才发现这里的字全是乱的。

人眼去阅读一段话的时候，会顾及上下文，不是依次单个字符的识别，因此引入循环神经网络去识别上下文能够极大提升模型的准确率。在决赛中，序列有几个地方都是有上下文关系的：

- 前面一个或两个赋值式一定是“中文 = 数字 ;”这样的形式。
- 左括号一定会有右括号。
- 括号的位置是有语法规则的。
- 一定会有一个分式。
- 分式的分子一定是中文。
- 如果只有一个赋值式，那么表达式中的中文一定是赋值式的中文。
- 如果有两个赋值式，赋值式容易看清，表达式不容易看清，那么可以通过赋值式的中文去修正表达式的中文，特别是分子中文被裁掉的时候。

3. 其他参数

相比之前初赛的模型，这里进行了一些修改：

- padding 变为 same，不然觉得特征图的高度不够，无法识别分数。
- 增加了 l2 正则化，loss loss 变得更大了，但是准确率变得更高了（添加 l2 的部分包括卷积层的 kernel，BN 层的 gamma 和 beta，以及全连接层的 weights 和 bias）。
- 各个层的初始化变为 he_uniform，效果比之前好。
- 去掉了 dropout，不清楚影响如何，但是反正有生成器，应该不会出现过拟合的情况。
- 修改过 GRU 的 implementation 为 2，原因是希望显卡能加快 GRU 的速度，但是似乎速度还不如设置为 0，使用 CPU 来运行，所以又改回来了。

l2 正则化的参数直接参考了 Xception 论文 5.3 节给的参数：

```
  Weight decay: The Inception V3 model uses a weight decay (L2 regularization)
rate of 4e-5, which has been carefully tuned for performance on ImageNet. We
found this rate to be quite suboptimal for Xception and instead settled for
1e-5.
```

10.4.5 生成器

为了得到更多的数据，提高模型的泛化能力，使用了一种很简单的数据扩充办法，就是根据表达式的中文随机挑选赋值式，组成新的样本。这里取了前 350*256=89600 个样本来生成，用之后的 10240 个样本进行验证集，还有一点零头因为太少就没有用了。

导入数据的时候，先读取运算式的图像，然后按照中文导入赋值式的图像到字典中。因为字典中的 key 是无序的，所以在字典中保存的是 list，列表是有序的。

```
    from collections import defaultdict

    cn_imgs = defaultdict(list)
    cn_labels = defaultdict(list)
    ss_imgs = []
    ss_labels = []

    for i in tqdm(range(n1)):
        ss = df[0][i].decode('utf-8').split(';')
        m = len(ss)-1
        ss_labels.append(ss[-1])
        ss_imgs.append(cv2.imread('crop_split2/%d_%d.png'%(i, 0)).
transpose(1, 0, 2))
        for j in range(m):
            cn_labels[ss[j][0]].append(ss[j])
            cn_imgs[ss[j][0]].append(cv2.imread('crop_split2/%d_%d.png'%(i,
m-j)).transpose(1, 0, 2))
```

接下来实现生成器，这里继承了 Keras 中的 Sequence 类：

```
    from keras.utils import Sequence

    class SGen(Sequence):
        def __init__(self, batch_size):
            self.batch_size = batch_size
            self.X_gen = np.zeros((batch_size, width, height, 3), dtype=np.
uint8)
            self.y_gen = np.zeros((batch_size, n_len), dtype=np.uint8)
            self.input_length = np.ones(batch_size)*rnn_length
            self.label_length = np.ones(batch_size)*38

        def __len__(self):
            return 350*256 // self.batch_size

        def __getitem__(self, idx):
            self.X_gen[:] = 0
            for i in range(self.batch_size):
                try:
                    random_index = random.randint(0, n1-1)
                    cls = []
                    ss = ss_labels[random_index]
                    cs = re.findall(ur'[\u4e00-\u9fff]',df[0][random_index].
decode('utf-8').split(';')[-1])
```

```
                    random.shuffle(cs)
                    x = 0
                    for c in cs:
                        random_index2 = random.randint(0, len(cn_
labels[c])-1)
                        cls.append(cn_labels[c][random_index2])
                        img = cn_imgs[c][random_index2]
                        w, h, _ = img.shape
                        self.X_gen[i, x:x+w, :h] = img
                        x += w+2
                    img = ss_imgs[random_index]
                    w, h, _ = img.shape
                    self.X_gen[i, x:x+w, :h] = img
                    cls.append(ss)

                    random_str = u';'.join(cls)
                    self.y_gen[i,:len(random_str)] = [characters.find(x) for
x in random_str]
                    self.y_gen[i,len(random_str):] = n_class-1
                    self.label_length[i] = len(random_str)
                except:
                    pass

            return [self.X_gen, self.y_gen, self.input_length, self.label_
length], np.ones(self.batch_size)
```

首先随机取一个表达式，然后用正则表达式查找里面的中文，再从“{ 中文：图像数组 }”的字典中随机取图像，经过之前预处理的方式拼接成一个新的序列。

例如，随机取了一个“85882*(河 /76020-37023)- 铁”，然后从铁的赋值式中随机取一个，再从河的赋值式中随机取一个，拼起来就能得到图 10-30。

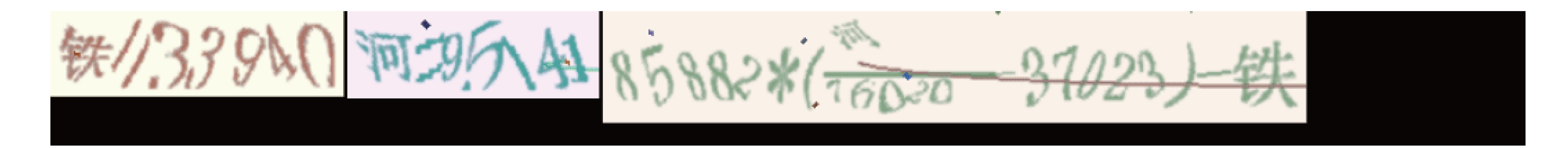


图 10-30　数据增强可视化

可以看到背景颜色是不同的，但是并不影响模型去识别。

10.4.6　模型的训练

训练的策略是先用 Adam() 默认的学习率 1e-3 快速收敛 50 代，然后用 Adam(1e-4) 运行 50 代，达到一个不错的 loss，最后用 Adam(1e-5) 微调 50 代，每一代都保存权值，并且把验证集的准确率运行出来。图 10-31 中绿色的线 0.9977 就是按照上面的方法训练的模型。

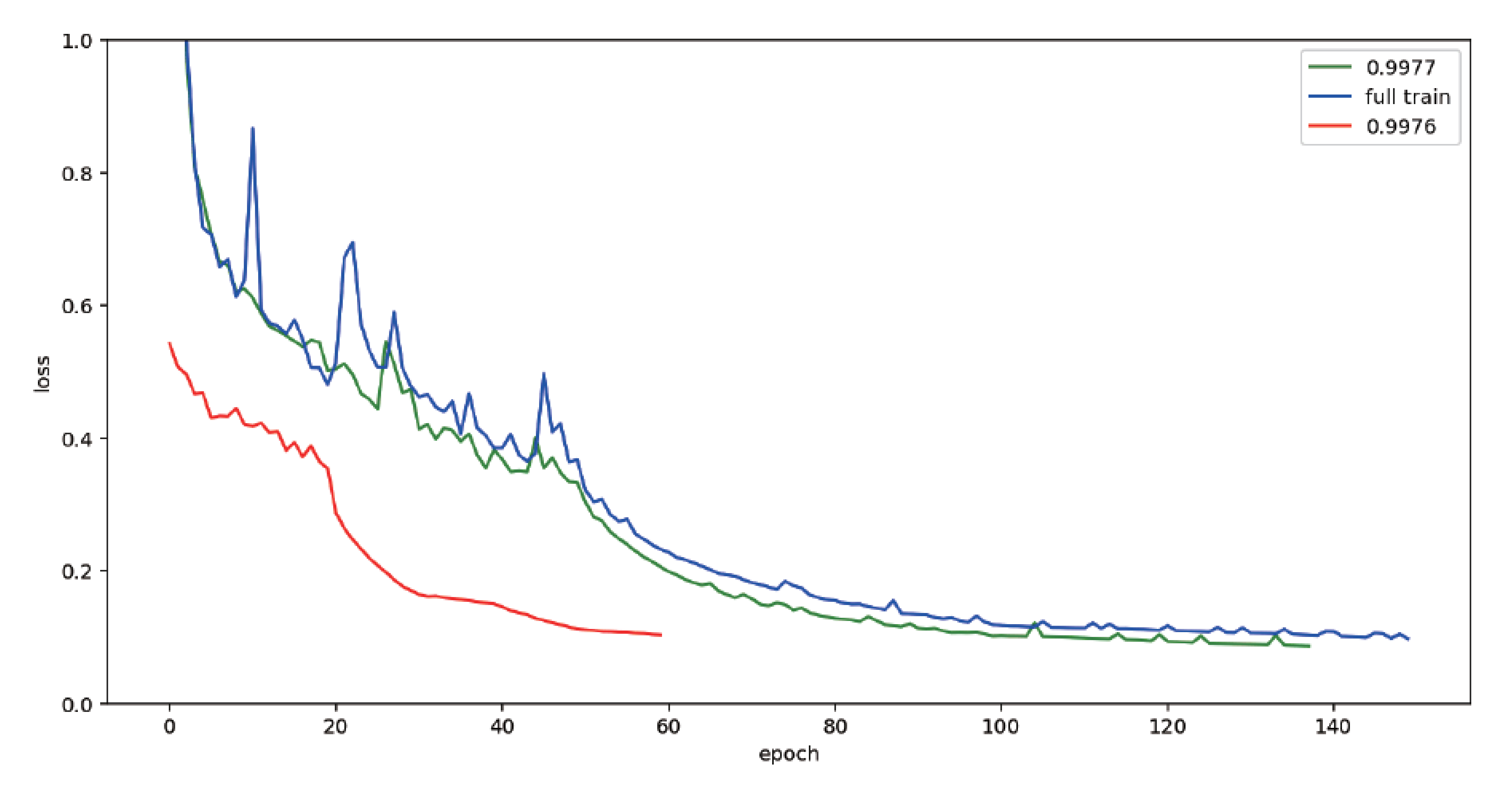


图 10-31 训练过程 loss 可视化

当然，还尝试过先按照 1e-3 的学习率训练 20 代，然后 1e-4 和 1e-5 交替训练 2 次，每次训练取验证集 loss 最低的结果继续训练，也就是图 10-31 中红色的线，虽然速度快，但是准确率不够好。

之后将全部训练集用于训练，得到蓝色的线，效果和绿色差不多。

10.4.7 预测结果

读取测试集的样本，然后用 base_model 进行预测。这个过程很简单，就不赘述了。

```
    X = np.zeros((n, width, height, channels), dtype=np.uint8)

    for i in tqdm(range(n)):
        img = cv2.imread('crop_split2_test/%d.png'%i).transpose(1, 0, 2)
        a, b, _ = img.shape
        X[i, :a, :b] = img

    base_model = load_model('model_346_split2_3_%s.h5' % z)
    base_model2 = make_parallel(base_model, 4)

    y_pred = base_model2.predict(X, batch_size=500, verbose=1)
    out = K.get_value(K.ctc_decode(y_pred[:,2:], input_length=np.ones(y_
pred.shape[0])*rnn_length)[0][0])[:, :n_len]
```

输出到文件的部分有一点值得一提，就是如何计算出真实值：

```
    ss = map(decode, out)

    vals = []
    errs = []
    errsid = []
```

```
    for i in tqdm(range(100000)):
        val = ''
        try:
            a = ss[i].split(';')
            s = a[-1]
            for x in a[:-1]:
                x, c = x.split('=')
                s = s.replace(x, c+'.0')
            val = '%.2f' % eval(s)
        except:
    #           disp3(i)
            errs.append(ss[i])
            errsid.append(i)
            ss[i] = ''

        vals.append(val)

    with open('result_%s.txt' % z, 'w') as f:
        f.write('\n'.join(map(' '.join, list(zip(ss, vals)))).
encode('utf-8'))

    print len(errs)
    print 1-len(errs)/100000.
```

运算结果：

```
# output
22
0.99978
```

其中的思路说起来很简单，就是将表达式中的赋值式中文替换为赋值式的数字，然后直接用 python eval 得到结果，算不出来的直接留空即可。这个 0.9977 模型的可算率达到了 0.99978，也就是说十万个样本中只有 22 个样本不可算。当然，实际上还是有一些样本即使可算，也会因为各种原因识别错，例如 5 和 6 就是错误的重灾区，某些数字被干扰线切过，导致肉眼都辨认不清等。

10.4.8 模型结果融合

模型结果融合的规则很简单，对所有的结果进行次数统计，首先去掉空的结果，然后取最高次数的结果即可，其实就是简单的投票。

```
    import glob
    import numpy as np
    from collections import Counter

    def fun(x):
```

```
        c = Counter(x)
        c[' '] = 0
        return c.most_common()[0][0]

    ss = [open(fname, 'r').read().split('\n') for fname in glob.
glob('result_model*.txt')]
    s = np.array(ss).T
    with open('result.txt', 'w') as f:
        f.write('\n'.join(map(fun, s)))
```

将上面 loss 图中的三个模型结果融合，最后得到了 0.99868 的测试集准确率。

10.4.9 其他尝试

1. 不定长图像识别

在比赛刚开始的时候，尝试过将图像的宽度设置为 None，也就是不定长的宽度，但是由于无法解决 reshape 的问题，这个方案被否了。

2. 分别识别

之前尝试过图像切成几块分别识别，赋值式和表达式的模型分开，考虑到无法得到上下文的信息，可能会丢失一定的准确率，做到一半否掉了这个方案。

3. 生成器尝试

我们尝试过写一个生成器，但是由于和官方给的图像差太远，并且实际测试的时候要么是生成的准确率高、官方的准确率低，要么反过来，所以没有投入使用。

图 10-32 中第一个是官方的图像，后面 5 个是生成器生成的，可以看到我们的字没有官方的紧凑，等号也不太一样，而分式的字又太紧凑了。



图 10-32 生成器生成的图像

4. 其他 CNN 模型的尝试

除了自己搭模型，还尝试过用 ResNet、DenseNet 替换 CNN，然后进行训练。但是这些模型本身就很大，训练起来速度很慢，前面的 val_loss 一直在抖动，并且最终提交的效果又和浅层模型没有太大差别，所以为了快速尝试更多方案，舍弃了类似 ResNet 复杂模型（见图 10-33）的想法。

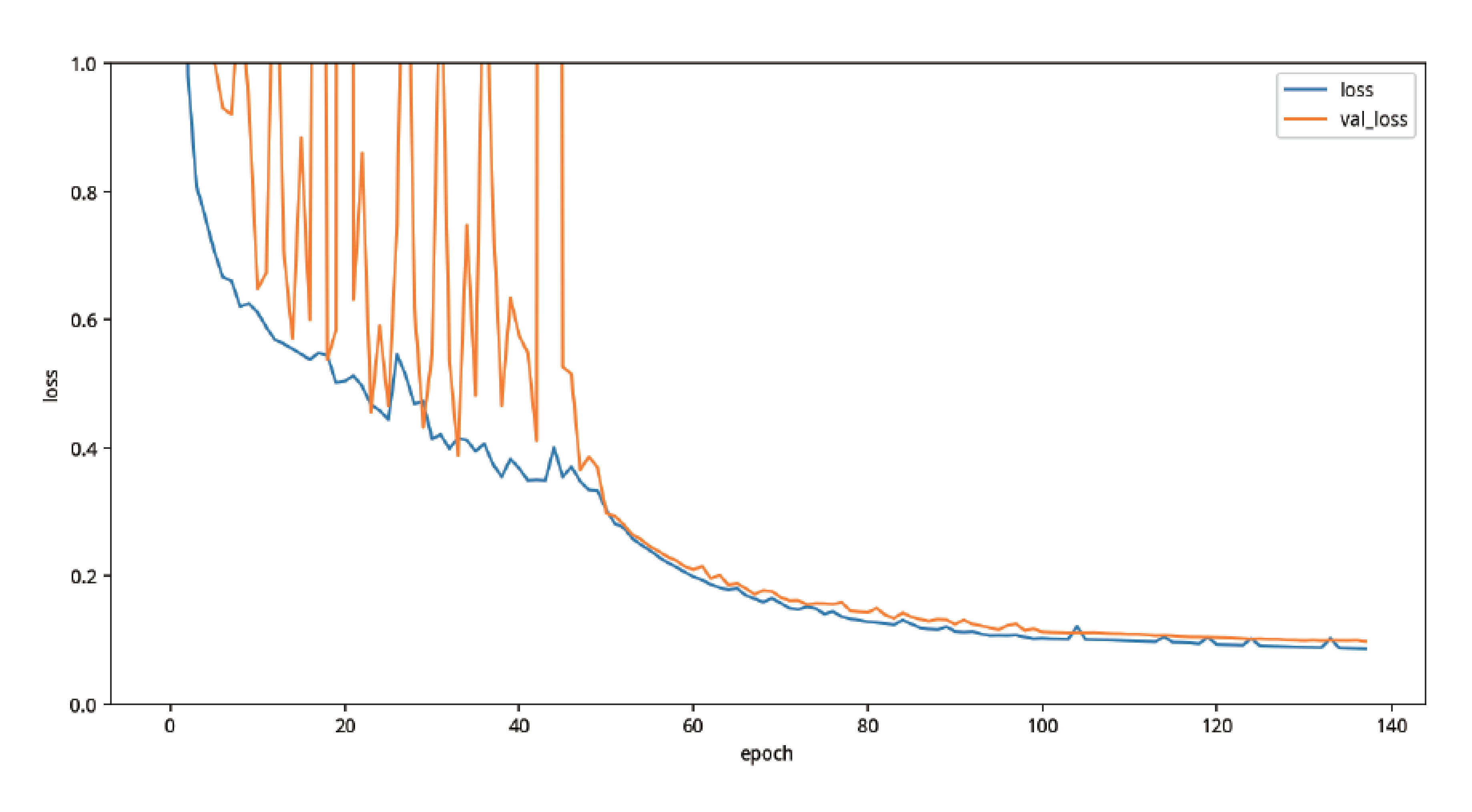


图 10-33　复杂模型训练过程 loss 可视化

5. 替换 GRU 为 LSTM

在比赛最后尝试过将 GRU 替换为 LSTM，得到的结果是十分类似的（见图 10-34），但是提交上去后准确率有轻微下降（多错了几个样本，可能是运气问题），之前做验证码识别的时候也是替换过，效果差不多，因此没有继续尝试。理论上这个序列长度并没有很长，GRU 和 LSTM 影响不大。

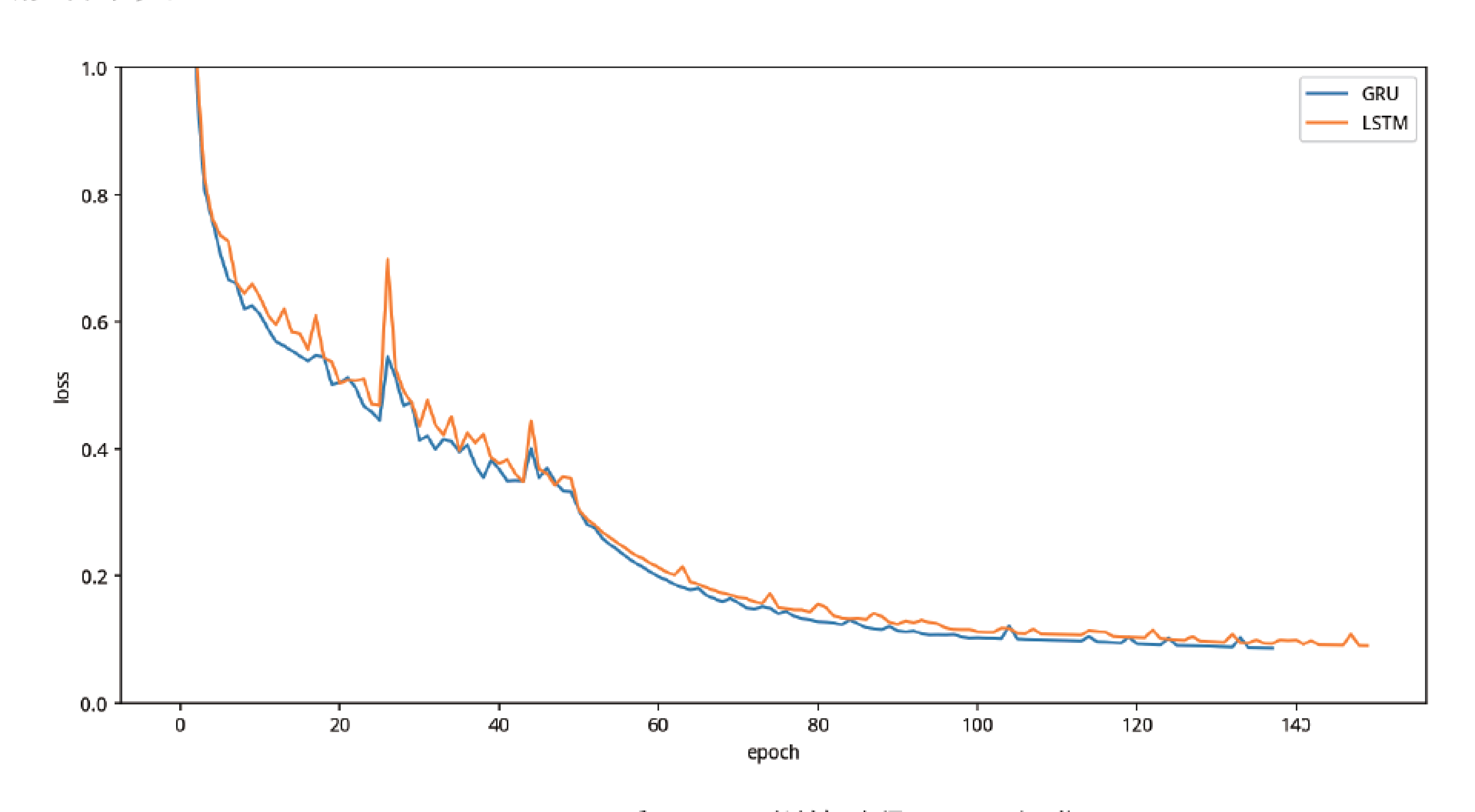


图 10-34　GRU 和 LSTM 训练过程 loss 可视化

10.4.10 小结

1. 对项目的思考

本项目需要注意以下一些要点。

- 数据准备
  - 深度学习与传统图像处理技术相结合，可以达到更好的准确率。
  - 文本识别可以构造验证码生成器进行数据增强，增加训练样本数。
- 模型优化
  - 如何根据项目特点对模型结构进行调整，如 CNN 部分减少池化层使用等。
  - 为了防止过拟合，在模型中引入 L2 正则化。
- 模型训练
  - 使用学习率衰减策略，训练模型。
  - 对复杂的模型，可以将同一批次输入数据分摊给多个 GPU 进行计算。

2. 有趣的样本

在测试集里有一个 95170.png 样本很难分割，如图 10-35 所示。

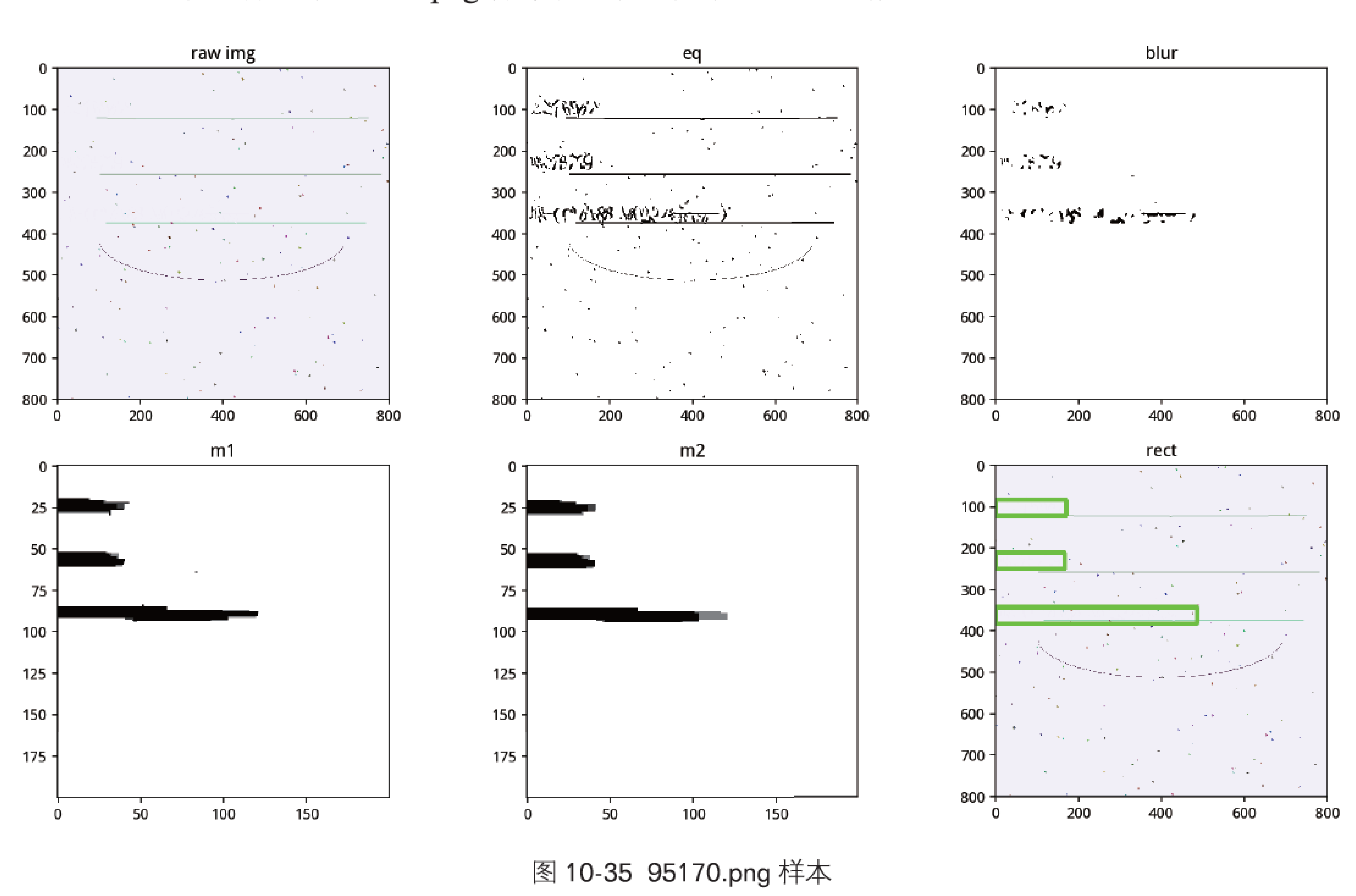


图 10-35 95170.png 样本

因为它的字太浅了，很难被切割出来，肉眼也基本上无法分辨。

它的表达式也很难切，稍有不慎就切割掉中文了，如图 10-36 所示。

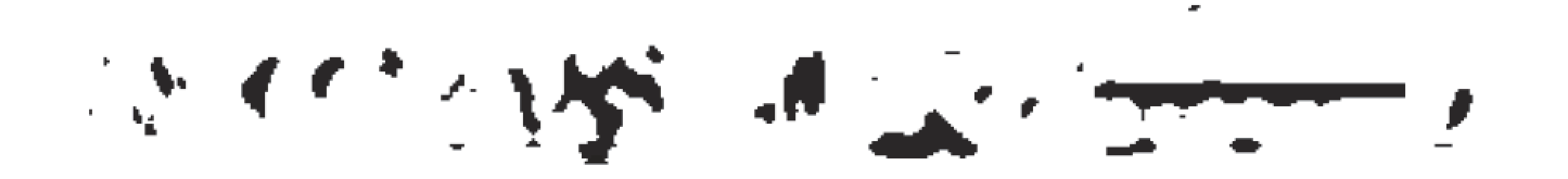


图 10-36　切割掉中文

在分割的验证集中，发现了被干扰线成功干扰的样本，如图 10-37 所示。

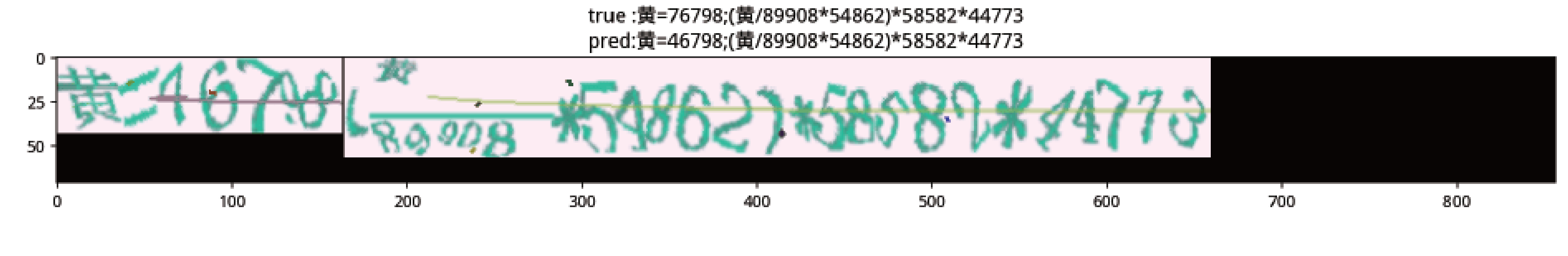


图 10-37　被干扰线干扰的样本

我们可以看到第一个“7”倾斜后加上一条干扰线，很容易就被模型辨别为“4”了，但是人类却不会犯这样的错，这也是 CNN 和人之间的区别，目测卷积层自动把图像转灰度图了。

3. 可能的改进

将生成器写出来，可以获取无穷无尽的样本。对 5 和 6 的识别，以及更多横向的中文的识别，会有很好的帮助。

做更好的预处理，比如干扰线和字的颜色是不同的，可以通过程序去除，切图可以更精准一些，可以极大地提高训练速度。

使用其他的模型，比如竞赛讨论群中有人提到的 attention 模型，或者看看 OCR 相关的论文，查找更多的模型融合结果，比直接运行类似结构的模型来融合的效果会好很多。

10.5 参考文献及网页链接

［1］ 三次简化一张图：一招理解 LSTM/GRU 门控机制 . 知乎专栏 . Available at: https://zhuanlan.zhihu.com/p/28297161.

［2］ OpenCV: Histograms - 2: Histogram Equalization. Available at: http://docs.opencv.org/master/d5/daf/tutorial_py_histogram_equalization.html.

［3］ OpenCV: Changing Colorspaces. Available at: http://docs.opencv.org/master/df/d9d/tutorial_py_colorspaces.html.

［4］ OpenCV: Morphological Transformations. Available at: http://docs.opencv.org/master/d9/d61/tutorial_py_morphological_ops.html.

［5］ OpenCV: Smoothing Images. Available at: http://docs.opencv.org/master/d4/d13/ tutorial_py_filtering.html.

［6］ OpenCV: Contour Features. Available at: http://docs.opencv.org/master/dd/d49/ tutorial_py_contour_features.html.

［7］ OpenCV: Image Thresholding. Available at: http://docs.opencv.org/master/d7/d4d/ tutorial_py_thresholding.html.

［8］ Amodei, D. et al. Deep Speech 2: End-to-End Speech Recognition in English and Mandarin. [1512.02595] Deep Speech 2: End-to-End Speech Recognition in English and Mandarin (2015). Available at: https://arxiv.org/abs/1512.02595.

［9］ Contour Approximation Method. OpenCV: Contours : Getting Started. Available at: http://docs.opencv.org/master/d4/d73/tutorial_py_contours_begin.html.

［10］ Fchollet. fchollet/keras. GitHub. Available at: https://github.com/fchollet/keras/blob/master/ examples/image_ocr.py.

［11］ Lepture. lepture/captcha. GitHub (2017). Available at: https://github.com/lepture/captcha.

［12］ baidu-research. baidu-research/warp-ctc. GitHub (2017). Available at: https://github.com/baidu-research/warp-ctc.

［13］ captcha. 使用深度学习来破解 captcha 验证码 . 知乎专栏 . Available at: https://zhuanlan.zhihu.com/p/26078299.

第 11 章

见习医生——使用全卷积神经网络分割病理切片中的癌组织

这里简单介绍如何使用第 5 章和第 6 章提到的全卷积神经网络（FCN），来预测病理切片中哪部分是癌症组织。这里通过对 560 张病理切片进行学习，得到了不错的预测结果，这个模型至少可以作为辅助诊断工具，减少病理医生的工作量。

11.1 任务描述

赛题来自数愿组织的病理切片 AI 识别挑战赛初赛赛题（www.datadreams.org/ race-race-3.html）。官方描述如下。

11.1.1 赛题设置

大赛选取胃癌病理切片图像为比赛数据，参赛团队运用人工智能的技术，开发算法模型，即通过胃癌病理切片数据，检测判断病理切片图像有无癌症。大赛通过探索胃癌病理切片智能

诊断的优秀算法，提升胃癌检测的效率，协助医生诊疗。参赛者可下载数据，在本地调试算法，提交结果由机器自动评测成绩，并定时公布排行榜。

11.1.2 数据描述

初赛选取胃癌病理切片，为常规 HE 染色，放大倍数 20×，图片大小为 2048×2048 像素，比赛数据为整体切片的部分区域，tiff 格式。比赛不允许使用外部数据。 初赛选取 100 个病人案例（部分为癌症、部分为非癌症），共计 1000 张病理切片图片，训练集数量 700 张，测试集数量 300 张。

11.1.3 数据标注

病理专家将数据标记（双盲评估 + 验证）为有无癌症，并用线条画出肿瘤区域轮廓。原始数据以及标注数据内容如图 11-1 所示。

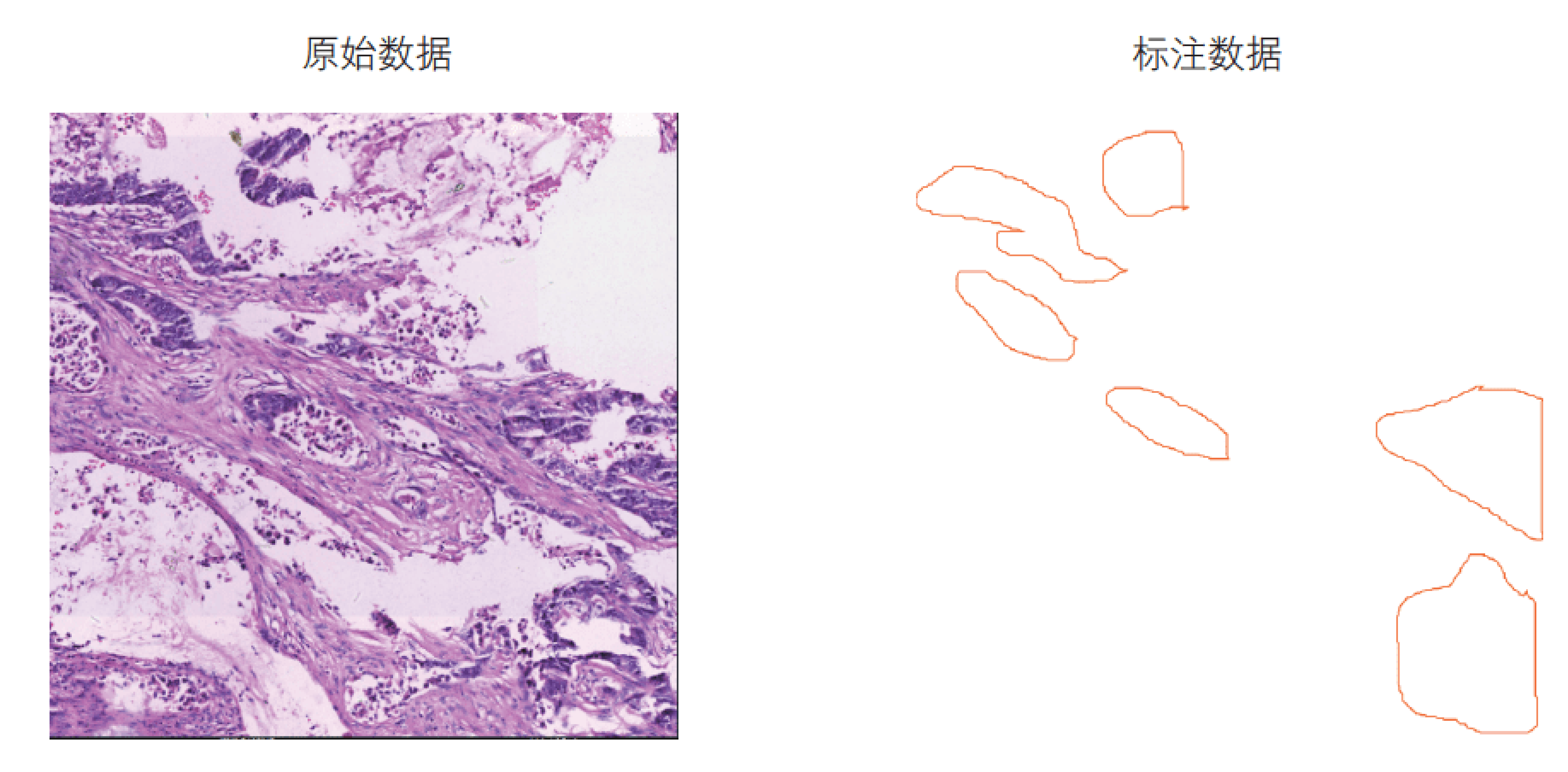


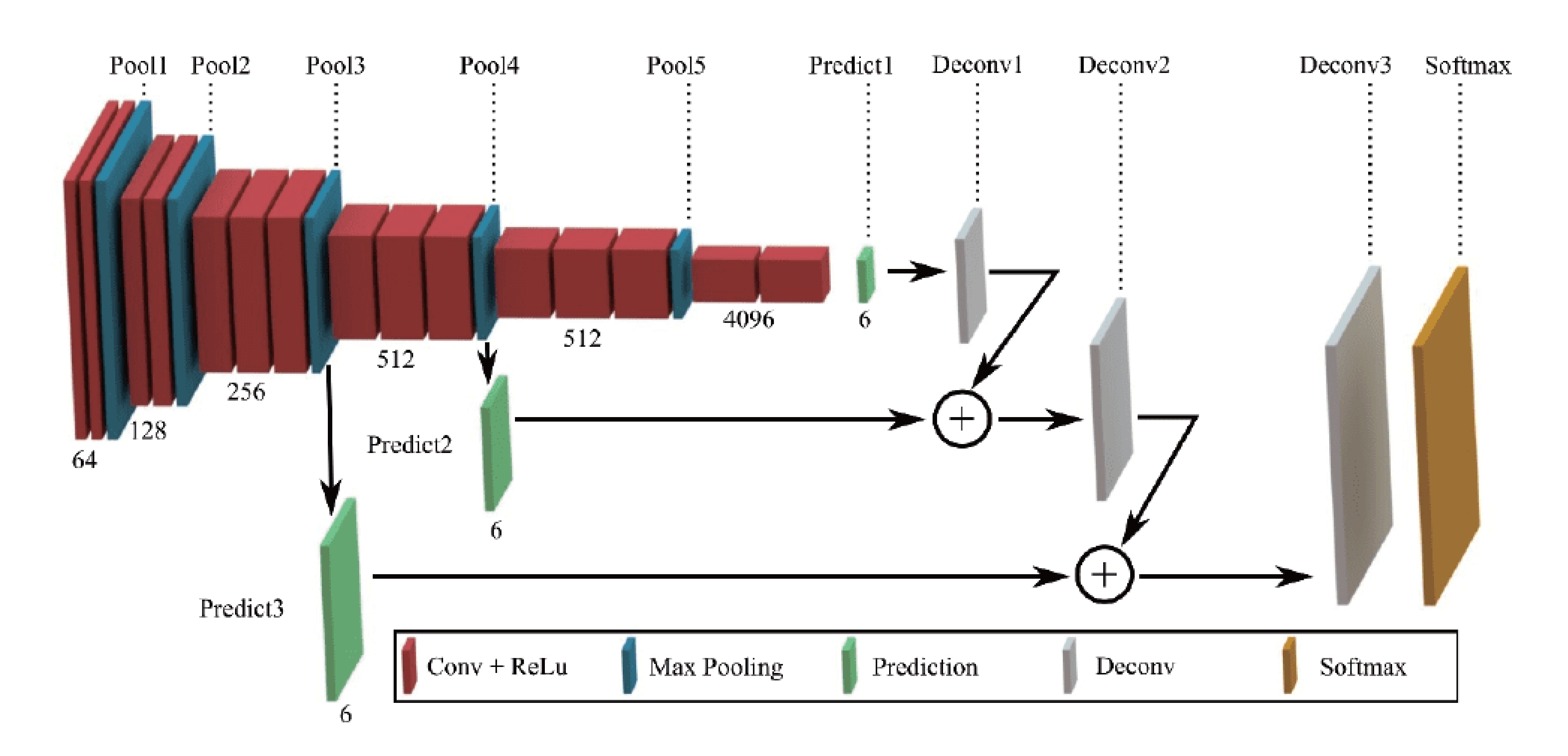
图 11-1 原始数据与标注数据

11.2 总体思路

由于需要实现像素级别的图像分割，因此考虑使用全卷积神经网络（FCN），使用如图 11-2 所示的架构。

在这种架构基础之上，有以下想法：

- 图形分辨率很高，可以考虑将一张图片拆分为多张小图片进行模型训练。
- 由于只有 700 张图片，并且是 RGB 图片，根据在第 9 章猫狗大战中的经验，以及最后提到的皮肤癌判断项目的分析架构，这里考虑引入迁移学习。
- 样本数量有限，可能会有过拟合现象发生，考虑在反卷积层中引入 l2 正则化。
- 样本数量有限，考虑进行数据增强。



（图片来源：PCA-aided Fully Convolutional Networks for Semantic Segmentation of Multi-channel fMRI）

图 11-2　全卷积神经网络架构

对于第一点，其实我们的训练图像已经是经过切割之后的图片了——病理切片的图片大小通常是上万个像素点 × 上万个像素点的分辨率，这里已经是切割成 2048×2048 的小图片了。这里的测试中，发现直接将 2048×2048 的输入图片转换成 256×256 分辨率后训练，单张 GPU 占用 10GB 显存的情况下，训练模型通常需要接近两小时的时间。因此，这里就不继续将图片拆成小图片了，直接将输入图片分辨率转换成 256×256，降低分辨率直接训练。读者如果在云服务器租用显存更高、显卡更多的机器，可以考虑将图片进行二次裁剪，剪成更多的图片，而不是简单地减小分辨率，进而用更高的分辨率训练模型。

对于第二点，可以直接将 VGG16 模型中 imageNet 的训练结果运用到这里，在此基础上，进一步地添加反卷积层。

对于第三点，引入正则化的话，同我们第 2 章逻辑回归类似，在引入 l2 正则化时，需要注意在损失函数处将模型的权重加进去。

对于第四点，由于病理切片不涉及拍摄的角度问题，可以引入旋转以及小范围的缩放，但由于癌症区域在颜色、形态、细小颗粒等特征会和正常区域有所区别，因此数据增强的过程中，不引入噪点，不改变颜色。

11.3　构造模型

由于这部分内容需要实现的功能比较复杂，不再是像之前一样的顺序执行（Sequencial）的架构，这里没有继续使用 Keras 框架。实际上，我们也没有必要直接用更底层的 tf.nn 模块，现有版本 TensorFlow（v1.1.0）的 tf.layers 模块使用起来也很方便。

模型编写仍然遵循三段论——准备数据、构建模型以及模型优化。

11.3.1 准备数据

准备数据中遇到的第一个问题，就是如何处理 svg 格式的标注数据。首先，svg 的标注，看起来是一个空心的区域，实际模型训练过程中，需要将其转换成实心区域。其次，svg 是一种矢量图，并不是以矩阵的形式存储的，需要将其转换为矩阵。

对于第一点，打开 svg 文件会发现这里标注区域使用了诸如 <polygon fill=” none” points= “537,742 ... 537,742” stroke=” #f8691c” stroke-width=” 5” /> 这样的形式。我们意识到图像空心的原因，是因为 fill 里面写的内容是 none，可以改成黑色，换成 fill=” #FFFFFF。注意，标注区域有橘黄色的边，填充色换成黑色的同时，也需要将多边形区域边的颜色设置为黑色。于是这里可以考虑使用正则表达式直接替换：

```
    svg_code = re.sub(r'''fill="(\w+|None)"''', '''fill="#FFFFFF"''', svg_code)
    svg_code = re.sub(r'''stroke="(\w+|None)"''', '''stroke="#FFFFFF"''',
svg_code)
```

然后对第二点，将修改后 svg 结果以 png 的格式保存下来：

```
    svg2png(bytestring=svg_code, write_to="out.png")
```

完整函数如下：

```
    def get_image(path, shape):
        """
        使用 opencv 读取图像
        :param  path  输入图像路径
        :param  shape 输出图像大小
        :return image 以 [H,W,C] 的 RGB 矩阵形式，输出图像
        """
        image = cv2.imread(path)
        image = image[:,:,::-1]
        if shape != None:
            image = cv2.resize(image, shape)
        return image

    def svg_process(svg_file, shape):
        """
        将癌细胞区域标注的矢量图 svg 格式的文件标注，转换成矩阵，并读入
        :param  svg_file  输入矢量图图像路径
        :param  shape      输出图像大小
        :return x          以灰度矩阵形式，输出癌细胞区域标注的结果
        """
        image_dir    = os.path.dirname(svg_file)
        image_prefix = os.path.basename(svg_file).split(".svg")[0]
```

```
        if not os.path.isfile("%s/%s.png" % (image_dir, image_prefix)):
            with open(svg_file, "r") as f_in:
                svg_code = f_in.readlines()
                svg_code = "".join(svg_code[1:])
                svg_code = re.sub(r'''fill="(\w+|None)"''',
'''fill="#FFFFFF"''', svg_code)
                svg_code = re.sub(r'''stroke="(\w+|None)"''',
'''stroke="#FFFFFF"''', svg_code)
                svg2png(bytestring=svg_code, write_to="%s/%s.png" % (image_
dir, image_prefix))

        img = get_image("%s/%s.png" % (image_dir, image_prefix), shape)
        return cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

这部分遇到的第二个问题是数据增强。首先考虑能否直接使用 Keras 的现成函数，如同第 7 章 CIFAR10 以及第 9 章猫狗大战中的代码一样。

但是，麻烦事来了，我们查阅 Keras API 后，发现 keras.preprocessing.image 的这个模块中，之前使用的 ImageDataGenerator 这个函数输出的是一个图像矩阵，外加一个分类种类的标签。而我们需要的是一个图像矩阵，以及一个经过同样旋转、缩放变换的标注矩阵。目前（V2.0.4 版）这个功能在 Keras 里并没能够直接实现。所以需要写一个可以同时旋转病理图像以及标注图像的函数。

附注

比较打脸的是，Keras API 虽然没有直接实现这个功能，但是官方文档中有提供如何使用 ImageDataGenerator 同时变换图像和标注区域的案例。具体而言，就是用两个参数相同的生成器，使用同一套随机种子，对不同文件夹的输入执行两次即可。在作者写后面的代码之后，考虑能否给 Keras 项目“贡献自己的一份力量”时，发现 Keras 官方文档给出了这样的方法：

```
    data_gen_args = dict(featurewise_center=True,
                         featurewise_std_normalization=True,
                         rotation_range=90.,
                         width_shift_range=0.1,
                         height_shift_range=0.1,
                         zoom_range=0.2)
    image_datagen = ImageDataGenerator(**data_gen_args)
    mask_datagen = ImageDataGenerator(**data_gen_args)

    # Provide the same seed and keyword arguments to the fit and flow methods
    seed = 1
    image_datagen.fit(images, augment=True, seed=seed)
    mask_datagen.fit(masks, augment=True, seed=seed)
```

```
image_generator = image_datagen.flow_from_directory(
    'data/images',
    class_mode=None,
    seed=seed)

mask_generator = mask_datagen.flow_from_directory(
    'data/masks',
    class_mode=None,
    seed=seed)

# combine generators into one which yields image and masks
train_generator = zip(image_generator, mask_generator)

model.fit_generator(
    train_generator,
    steps_per_epoch=2000,
    epochs=50)
```

所以接下来的内容，大家可以当成学习材料，实际运用中可以考虑使用更简单的 Keras 官方案例。

实现这个功能的方法有很多，这里采用的方法是，改造 Keras 功能接近的函数。keras.preprocessing.image 这个模块位于 https://github.com/fchollet/keras/blob/2.0.4/keras/ preprocessing/image.py，看起来相对独立，没有复杂的引用关系，因此可以简单地阅读下 Keras 如何在一张图片中实现图像反转、缩放，进而通过简单的改造实现在另一张图片中执行同样操作。

首先回忆第 8 章，我们是如何对 CIFAR-10 数据使用生成器的：

```
from keras.preprocessing.image import ImageDataGenerator


datagen = ImageDataGenerator(...)
datagen.fit(X_train)
```

于是在 ImageDataGenerator 类中的 fit 方法进一步定位到 augment，以及 random_transform 方法所在的位置：

```
# 649-654, fit
if augment:
    ax = np.zeros(tuple([rounds * x.shape[0]] + list(x.shape)[1:]),
dtype=K.floatx())
    for r in range(rounds):
        for i in range(x.shape[0]):
            ax[i + r * x.shape[0]] = self.random_transform(x[i])
```

```
        x = ax

    theta = np.pi / 180 * np.random.uniform(-self.rotation_range, self.
rotation_range)
    zx, zy = np.random.uniform(self.zoom_range[0], self.zoom_range[1], 2)

    # 566-570 random_transform
    if theta != 0:
        rotation_matrix = np.array([[np.cos(theta), -np.sin(theta), 0],
                                    [np.sin(theta), np.cos(theta), 0],
                                    [0, 0, 1]])
        transform_matrix = rotation_matrix

    # 584-588 random_transform
    if zx != 1 or zy != 1:
        zoom_matrix = np.array([[zx, 0, 0],
                                [0, zy, 0],
                                [0, 0, 1]])
        transform_matrix = zoom_matrix if transform_matrix is None else
np.dot(transform_matrix, zoom_matrix)

    # 593-594 random_transform
    x = apply_transform(x, transform_matrix, img_channel_axis,fill_mode=self.
fill_mode, cval=self.cval)
```

我们发现，Keras 这部分代码的逻辑是，首先从定义的范围内，根据均匀分布（Uniform），随机生成一个旋转角、缩放比，进而通过一个大小为 3×3 的转换矩阵（transform_matrix）来表示旋转、缩放。最终通过 apply_transform 函数实现图像的旋转、缩放。

那么我们的改造思路，就是根据定义的旋转、缩放范围，随机生成旋转角、缩放比之后，进而生成转换矩阵，再分别将这个转换矩阵应用在输入病理图片以及癌症区域标注图片上。代码如下：

```
    def apply_transform(x,
                        transform_matrix,
                        channel_axis=0,
                        fill_mode='constant',
                        cval=0.):
        """
    进行图像旋转。简化改写了：
        https://github.com/fchollet/keras/blob/master/keras/preprocessing/image.py

        :param  x                         输入需要旋转的图像向量
```

```
        :param  transform_matrix    图像旋转矩阵参数
        :param  channel_axis        哪一个维度代表图像的编号。对 [N,H,W,C]，N 是图像
编号，所以是 0
        :param  fill_mode           填充由于旋转造成的边缘空白的方式。
                                    可选 {'constant', 'nearest', 'reflect', 'wrap'}
        :param  cval                如果是 'constant' 填充，则在空白处填写什么内容

        :return x                   旋转后的图像
        """
        x = np.rollaxis(x, channel_axis, 0)
        final_affine_matrix = transform_matrix[:2, :2]
        final_offset = transform_matrix[:2, 2]

        # 对癌症样本，同时有病理切片以及癌症区域标注图片
        if cval is False:
            cha_img1 = scipy.ndimage.interpolation.affine_transform(
                    x[0], final_affine_matrix, final_offset,
             order=0, mode=fill_mode, cval=False
       )
            cha_img2 = scipy.ndimage.interpolation.affine_transform(
                    x[1], final_affine_matrix, final_offset,
             order=0, mode=fill_mode, cval=True
       )
            channel_images = [cha_img1, cha_img2]

        # 对非癌症样本，不存在癌症区域标注图片
        else:
            channel_images = [scipy.ndimage.interpolation.affine_transform(
                x_channel,
                final_affine_matrix,
                final_offset,
                order=0,
                mode=fill_mode,
                cval=cval) for x_channel in x]

        x = np.stack(channel_images, axis=0)
        x = np.rollaxis(x, 0, channel_axis + 1)
        return x


    def picture_argument(image_input, image_gt, rotate, zoom):
        """
        对输入的病理切片，进行旋转、缩放操作的图像增强，并生成经过同样旋转、缩放操作的标注
区域
```

```
        :param  image_input              输入的病理切片图像
        :param  image_gt                 输入的病理切片癌症标注区域图像
        :param  rotate                   对图像进行旋转的正负角度范围
        :param  zoom                     对图像进行缩放操作的放大缩小百分比
        :return image_input, image_gt    旋转、缩放后的病理切片图像，以及对应的癌症区
域标注
        """
        theta = np.pi / 180 * np.random.uniform(-rotate, rotate)
        rotation_matrix = np.array([[np.cos(theta), -np.sin(theta), 0],
                                    [np.sin(theta), np.cos(theta), 0],
                                    [0, 0, 1]])

        transform_matrix = rotation_matrix
        zx, zy = np.random.uniform(1-zoom, 1+zoom, 2)
        zoom_matrix = np.array([[zx, 0, 0],
                                [0, zy, 0],
                                [0, 0, 1]])
        transform_matrix = np.dot(transform_matrix, zoom_matrix)

        h, w = image_input.shape[0], image_input.shape[1]
    transform_matrix = transform_matrix_offset_center(
                    transform_matrix, h, w
    )
        image_input = apply_transform(image_input, transform_matrix, 2,
                                    fill_mode="constant", cval=255)

        image_gt = apply_transform(image_gt, transform_matrix, 2,
                                    fill_mode="constant", cval=False)
        return image_input, image_gt
```

大功告成。接下来将这些函数包装进生成器。由于训练样本、验证样本需要有各自的生成器，因此，这里在生成器函数 batch_gen 外面加一层，这样就可以给两组样本每组分配一个生成器。

```
    def gen_batch_func(l_sample, image_shape):
        """
            生成器函数，输入样本名称，每调用一次生成器，输出若干张病理切片图片以及对应的癌症区
域标注图片

        :param  l_sample        输入的病理切片图像样本名称，
                                对应的病理切片图像放在 ./FCN/image/merge 目录下
                                对应的病理切片癌症区域标注放在 ./FCN/labels 目录下
        :param  image_shape     输出图片的大小
        :return batch_gen       输出生成器，每调用一次生成器，输出若干张病理切片图片以
及对应的癌症区域标注图片
        """
```

```
    def batch_gen(batch_size, augmentation=True):
        random.shuffle(l_sample)
        for batch_i in range(0, len(l_sample), batch_size):
            l_images = []
            l_gt_images = []
            for sample in l_sample[batch_i:batch_i+batch_size]:
                gt_image_file = "./FCN/labels/%s.svg" % (sample)
                image_file = "./FCN/image/merge/%s.tiff" % (sample)
                if os.path.isfile(gt_image_file):
                    gt_image_raw = svg_process(gt_image_file,
                                       shape=image_shape
                )

                else:
                    gt_image_raw = np.zeros(
                      [image_shape[0], image_shape[1]]
                )

                image = get_image(image_file, shape=image_shape)
                gt_image = gt_image_raw>100
                gt_image = gt_image.reshape(*gt_image.shape, 1)
                gt_image2 = np.bitwise_not(gt_image)
                gt_out = np.concatenate(
                        (gt_image, gt_image2), axis=2
         )
                if augmentation:
                    rotation = 180
                    zoom = 0.2
                    image,gt_out = picture_argument(
                      image, gt_out, rotation, zoom
                )

                l_images.append(image)
                l_gt_images.append(gt_out)

            yield np.array(l_images), np.array(l_gt_images)

    return batch_gen
```

11.3.2 构建模型

模型的构建阶段主要分为两个部分。

（1）第一部分是导入用 ImageNet 预训练的 VGG16 模型。这里如果直接使用官方地址的 ckpt 格式的模型，就需要首先将其转换成 .pb 格式的模型，并将 ckpt 中的参数值写入 .pb 文件。

这样做是因为直接加载 .pb 格式的模型相对容易，ckpt 格式只有参数，没有图的定义，需要在代码中定义模型结构，或者导入其他 .pb 文件。而 .pb 格式可以只存储图的模型，也可以进一步通过 TensorFlow 的 freeze_graph 功能，将参数写入 .pb 文件。我们这里为了省事，直接下载转换完成后的 vgg16.pb 文件。

```
    def load_vgg(sess, vgg_path):
        """
        载入 VGG16 预训练模型，返回我们基于 VGG16 训练全卷积神经网络 (FCN) 所必需的中间变量。
        :param  sess:     TensorFlow Session
        :param  vgg_path: vgg16 模型文件的下载路径。模型使用 pb 格式存储，
                          下载地址：https://s3-us-west-1.amazonaws.com/udacity-
selfdrivingcar/vgg.zip

        :return image_input, keep_prob, layer3_out, layer4_out, layer7_out
                         返回我们基于 VGG16 训练全卷积神经网络 (FCN) 所必需的中间变量
        """
        vgg_tag = 'vgg16'
        vgg_input_tensor_name = 'image_input:0'
        vgg_keep_prob_tensor_name = 'keep_prob:0'
        vgg_layer3_out_tensor_name = 'layer3_out:0'
        vgg_layer4_out_tensor_name = 'layer4_out:0'
        vgg_layer7_out_tensor_name = 'layer7_out:0'

        tf.saved_model.loader.load(sess, [vgg_tag], vgg_path)
        graph = tf.get_default_graph()
        input_image = graph.get_tensor_by_name(vgg_input_tensor_name)
        keep_prob = graph.get_tensor_by_name(vgg_keep_prob_tensor_name)
        vgg_layer3_out = graph.get_tensor_by_name(vgg_layer3_out_tensor_name)
        vgg_layer4_out = graph.get_tensor_by_name(vgg_layer4_out_tensor_name)
        vgg_layer7_out = graph.get_tensor_by_name(vgg_layer7_out_tensor_name)

        return input_image, keep_prob, vgg_layer3_out, vgg_layer4_out, vgg_
layer7_out
```

（2）第二部分是在 VGG16 模型的基础上，构建全卷积神经网络。这里根据 fully convolutional networks for semantic segmentation 这篇文章给定的网络结构，直接构建模型。这里需要注意的是，首先参数需要合理地初始化，我们在第 5 章介绍卷积层时，提到卷积层的初始化过程中，要注意随着层数的增多，随机初始化引入的方差，会随着连续的乘法运算，累计增加或者减少，进而影响整个梯度的计算。因此这里同样需要注意参数的合理初始化，这里引入了 xavier_initializer()。其次可以将卷积核通过 tf.slice 抽出来作为灰度图像，通过 tf.summary.image() 留下记录，这样就可以在 tensorboard 中看见卷积核的结果。深度学习虽然不容易解释，被人当作“玄学”，但实际上并非无法解释，通过对卷积核进行可视化分析，会给用户提供很多有用信息。

```
def layers(vgg_layer3_out, vgg_layer4_out, vgg_layer7_out, num_classes):
    """
    基于 load_vgg 返回的 VGG16 模型中间结果，设计全卷积神经网络（FCN）模型
    :param vgg_layer7_out: TF Tensor for VGG Layer 3 output
    :param vgg_layer4_out: TF Tensor for VGG Layer 4 output
    :param vgg_layer3_out: TF Tensor for VGG Layer 7 output
    :param num_classes: 需要分类的种类数目。这里是肿瘤区域 / 非肿瘤区域的二分类
    :return: 全卷积神经网络模型（FCN）的输出结果
    """
    with tf.name_scope("32xUpsampled") as scope:
        conv7_1x1 = tf.layers.conv2d(
  vgg_layer7_out, num_classes, 1,
  padding='same', name="32x_1x1_conv",
  kernel_regularizer=tf.contrib.layers.l2_regularizer(g_l2),
          kernel_initializer=tf.contrib.layers.xavier_initializer()
  )
        conv7_2x  = tf.layers.conv2d_transpose(
  conv7_1x1, num_classes, 4,
  strides=2, padding='same', name="32x_conv_trans_upsample",
  kernel_regularizer=tf.contrib.layers.l2_regularizer(g_l2),
  kernel_initializer=tf.contrib.layers.xavier_initializer()
)

    with tf.name_scope("16xUpsampled") as scope:
        conv4_1x1 = tf.layers.conv2d(
          vgg_layer4_out, num_classes, 1,
          padding='same', name="16x_1x1_conv",
          kernel_regularizer=tf.contrib.layers.l2_regularizer(g_l2),
          kernel_initializer=tf.contrib.layers.xavier_initializer()
        )
        conv_merge1 = tf.add(conv4_1x1, conv7_2x,
                          name="16x_combined_with_skip"
  )
        conv4_2x  = tf.layers.conv2d_transpose(
  conv_merge1, num_classes, 4,
          strides=2, padding='same',name="16x_conv_trans_upsample",
          kernel_regularizer=tf.contrib.layers.l2_regularizer(g_l2),
          kernel_initializer=tf.contrib.layers.xavier_initializer()
)

    with tf.name_scope("8xUpsampled") as scope:
        conv3_1x1 = tf.layers.conv2d(
          vgg_layer3_out, num_classes, 1,
          padding='same', name="8x_1x1_conv",
```

```
            kernel_regularizer=tf.contrib.layers.l2_regularizer(g_l2),
            kernel_initializer=tf.contrib.layers.xavier_initializer()
          )
          conv_merge2 = tf.add(conv3_1x1, conv4_2x,
           name="8x_combined_with_skip"
    )
          conv3_8x  = tf.layers.conv2d_transpose(
    conv_merge2, num_classes, 16,
            strides=8, padding='same', name="8x_conv_trans_upsample",
            kernel_regularizer=tf.contrib.layers.l2_regularizer(g_l2),
            kernel_initializer=tf.contrib.layers.xavier_initializer()
    )

     conv_image_0 = tf.slice(conv3_8x, [0,0,0,0], [-1,-1,-1,1])
     tf.summary.image("conv3_8x_results_0", conv_image_0)
     return conv3_8x
```

tf.summary.image 中的结果，训练过程中可以通过 tensorboard 进行可视化，结果类似图 11-3。

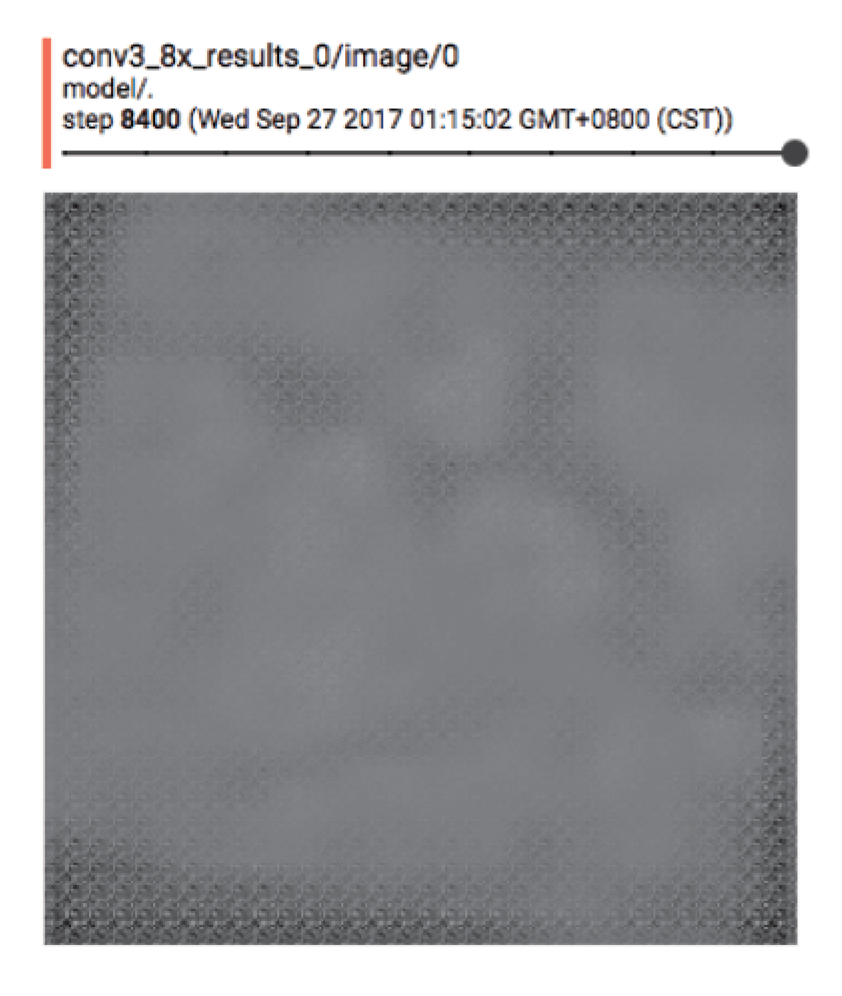


图 11-3　可视化的结果

11.3.3　模型优化

有目标，才有优化。之前我们做图像分类，目标就是尽可能多地将图片的种类分对。而现在我们做单个像素的图像分割，目标就是尽可能多地将图片中每一个像素的内容都分对。这个目标可以用交叉熵来代表，写成程序就是：

```
entropy_val = tf.nn.softmax_cross_entropy_with_logits(
          labels=true_label, logits=pred_label
)
cross_entropy_loss = tf.reduce_sum(entropy_val)
```

不过，在最开始的总体思路中，强调了这里由于样本少，需要引入正则化，防止过拟合。所以我们的目标要再加一条，就是在尽可能多地将图片中每一个像素的内容都分对的基础上，尽可能使用更少、更小的参数，防止参数过多造成对数据的过度迎合。这个升级版的目标，写成程序就是：

```
# 收集反卷积层用到的参数
reg_losses = tf.get_collection(tf.GraphKeys.REGULARIZATION_LOSSES)
# 更新的目标函数，将所有参数的平方和纳入优化目标
loss = cross_entropy_loss +  sum(reg_losses)
```

除了自己定义的目标以外，我们也需要考虑这项比赛的指标—— F1 值。比赛官网定义如下：

```
Evaluation of Cancerous Region Segmentation:
Precision=(|TP|)/(|TP|+|FP|)
Recall=(|TP|)/(|TP|+|FN|)
F1 Score=(2·Precision·Recall)/(Precision+Reall)
```

- TP：True Positive，被分类为属于癌变区域，但分类正确。
- FP：False Positive，被分类为属于癌变区域，但分类错误。
- FN：False Negative，被分类为属于非癌变区域，但分类错误。

于是根据官方的定义，我们写一个 F1 值的算式，作为一个独立于优化指标的另一项评价指标（这种独立性类似于第 2 章最后提到的 AUC 值）：

```
argmax_p = tf.argmax(pred_label, 1)
argmax_y = tf.argmax(true_label, 1)
TP = tf.count_nonzero( argmax_p   * argmax_y,    dtype=tf.float32)
TN = tf.count_nonzero((argmax_p-1)*(argmax_y-1), dtype=tf.float32)
FP = tf.count_nonzero( argmax_p   *(argmax_y-1), dtype=tf.float32)
FN = tf.count_nonzero((argmax_p-1)* argmax_y,    dtype=tf.float32)
precision = TP / (TP+FP)
recall    = TP / (TP+FN)
f1 = 2 * precision * recall / (precision + recall)
```

完整代码如下：

```
def optimize(nn_last_layer, correct_label, learning_rate, num_classes,
batch_size, split_idx):
        """
        定义模型的优化目标（损失函数），设置优化器
        :param  nn_last_layer:    全卷积神经网络模型（FCN）的输出结果
        :param  correct_label:    病理切片对应的、准确的癌症区域标注
        :param  learning_rate:    初始学习率大小
        :param  num_classes:      需要分类的种类数目。这里是癌症区域 / 非癌症区域的二分类
        :return pred_label:       病理切片对应的、模型预测的癌症区域标注
```

```
    :return training_op:      优化器
    :return cross_entropy_loss      交叉熵损失函数
    :return f1                            比赛规定的评价指标 f1 值
    :return learning_rate2          随训练次数逐步衰减后的学习率的大小
    """
pred_label = tf.reshape(
         nn_last_layer, [-1, num_classes], name="predicted_label"
)
true_label = tf.reshape(
         correct_label, [-1, num_classes], name="true_label"
)

    with tf.name_scope("f1_score"):
        argmax_p = tf.argmax(pred_label, 1)
        argmax_y = tf.argmax(true_label, 1)
        TP = tf.count_nonzero(
         argmax_p   * argmax_y,    dtype=tf.float32
  )
        TN = tf.count_nonzero(
         (argmax_p-1)*(argmax_y-1), dtype=tf.float32
  )
        FP = tf.count_nonzero(
         argmax_p   *(argmax_y-1), dtype=tf.float32
  )
        FN = tf.count_nonzero(
         (argmax_p-1)* argmax_y,    dtype=tf.float32
  )
        precision = TP / (TP+FP)
        recall    = TP / (TP+FN)
        f1 = 2 * precision * recall / (precision + recall)

    with tf.name_scope("cross_entropy_loss"):
        entropy_val = tf.nn.softmax_cross_entropy_with_logits(
         labels=true_label, logits=pred_label
  )
        cross_entropy_loss = tf.reduce_sum(entropy_val)
        reg_losses = tf.get_collection(
         tf.GraphKeys.REGULARIZATION_LOSSES
  )
        loss = cross_entropy_loss +  sum(reg_losses)

    with tf.name_scope("train"):
        batch = tf.Variable(0, tf.float32)
```

```
            learning_rate2 = tf.train.exponential_decay(
                  learning_rate,        # Base learning rate.
                  batch * batch_size,   # Current index into the dataset.
                  split_idx,            # Decay step.
                  0.95,                 # Decay rate.
                  staircase=True
       )
            # 不使用 learning_rate decay 策略的话，直接用 learning_rate
            #optimizer = tf.train.AdamOptimizer(learning_rate)

            optimizer = tf.train.AdamOptimizer(learning_rate2)
            training_op = optimizer.minimize(loss, global_step=batch)

        return pred_label, training_op, cross_entropy_loss, f1, learning_rate2
```

optimizer 函数作用于训练集，该函数输入了现有模型预测的结果以及真实结果，返回了对训练集数据表现的评价（交叉熵以及 f1 值），最后提出对整个模型参数的优化。对于验证集，同样可以得到现有模型对于验证集的预测结果以及验证集真实结果，但我们只需要对这个结果进行评价，不可以用验证集优化模型，因此对验证集的评价方法可以写成：

```
    def validation(nn_last_layer, correct_label, num_classes):
        """
        每当模型遍历所有训练样本（80%，560 个）之后，对剩下 20% 的验证样本执行一次验证操作，
检验模型在新样本上的表现
        :param  nn_last_layer:          全卷积神经网络模型（FCN）的输出结果
        :param  correct_label:          病理切片对应的、准确的癌症区域标注
        :param  num_classes:            需要分类的种类数目。这里是癌症区域 / 非癌症区
域的二分类
        :return cross_entropy_loss_cv   交叉熵损失函数（验证样本）
        :return f1_cv                   比赛规定的评价指标 f1 值（验证样本）
        """
    pred_label = tf.reshape(nn_last_layer, [-1, num_classes],
                            name="predicted_label_cv"
    )
    true_label = tf.reshape(correct_label, [-1, num_classes],
                            name="true_label_cv"
    )



        with tf.name_scope("f1_score_cv"):
            argmax_p = tf.argmax(pred_label, 1)
            argmax_y = tf.argmax(true_label, 1)
            TP = tf.count_nonzero(
```

```
            argmax_p   * argmax_y,    dtype=tf.float32
    )
        TN = tf.count_nonzero(
            (argmax_p-1)*(argmax_y-1), dtype=tf.float32
    )
        FP = tf.count_nonzero(
            argmax_p   *(argmax_y-1), dtype=tf.float32
    )
        FN = tf.count_nonzero(
            (argmax_p-1)* argmax_y,    dtype=tf.float32
    )
        precision = TP / (TP+FP)
        recall    = TP / (TP+FN)
        f1_cv = 2 * precision * recall / (precision + recall)

    with tf.name_scope("cross_entropy_loss_cv"):
        entropy_val = tf.nn.softmax_cross_entropy_with_logits(
            labels=true_label, logits=pred_label
    )
        cross_entropy_loss_cv = tf.reduce_sum(entropy_val)

    return cross_entropy_loss_cv, f1_cv
```

我们明确了优化目标之后，还需要做的一件事就是身体力行地优化这个模型。这部分内容在第 8 章的 Keras 版本中，只用 model.fit 一行就完成了。但是这里情况比较复杂，首先我们进来的数据并不是常见的分类数据，其次优化目标考虑 l2 正则化之后也更加复杂。因此这里还是要多写几行，实现：

- 多个 eproch 训练。
- 每个 eproch 中，每次用 batch_size 个样本，进行模型训练优化参数。
- 将这次训练的交叉熵以及 f1 值，写入 tensorboard 日志文件。
- 完成一个 eproch 的训练后，用验证集检验训练模型的表现（交叉熵 /f1 值）。

函数实现如下：

```
def train_nn(sess, epochs, batch_size, get_batches_train,
get_batches_cv, train_op, cross_entropy_loss,
             cross_entropy_loss_cv, f1, f1_cv,lr, input_image,
correct_label, keep_prob, learning_rate):
    """
    汇总之前的结果，训练定义的全卷积神经网络
    :param sess:                  TF Session
    :param epochs:                训练几轮数据
```

```
        :param batch_size:                批次大小
        :param get_batches_train:         获取训练数据的生成器，使用方法 gen_batch_
func(batch_size)
        :param get_batches_cv:            获取验证数据的生成器，使用方法 gen_batch_
func(batch_size)
    :param train_op:                      训练模型的操作子,
                            优化目标 cross_entropy_loss+l2_loss 最小化
        :param cross_entropy_loss:             交叉熵损失函数（训练样本）
        :param cross_entropy_loss_cv:          交叉熵损失函数（验证样本）
        :param f1:                             比赛规定的评价指标 f1 值（训练样本）
        :param f1_cv:                          比赛规定的评价指标 f1 值（验证样本）
        :param input_image:                    模型输入图片大小
        :param correct_label:                  病理切片对应的、准确的癌症区域标注
        :param keep_prob:                      VGG 模型中间参数
        :param learning_rate:                  初始化学习率大小

        """
        #save training results for every eproch
        saver = tf.train.Saver()
        model_dir = './usingNonAug_models_l2_norm_ExpDecay_lr_%1.2e_
l2_%1.2e_ e10_batch_%d' % (g_lr, g_l2, g_batch_size)
        log_dir   = "./usingNonAug_logs_l2_norm_ExpDecay_lr_%1.2e_l2_%1.2e_
e10_batch_%d"  % (g_lr, g_l2, g_batch_size)
        cv_dir    = "./usingNonAug_cv_ExpDecay_lr_%1.2e_l2_%1.2e_e10_batch_
%d.csv"  % (g_lr, g_l2, g_batch_size)
        if not os.path.isdir(model_dir):
            os.mkdir(model_dir)

        if not os.path.isdir(log_dir):
            os.mkdir(log_dir)

        f_out = open(cv_dir, "w")
        f_out.write("Eproch,cv_CrossEntropy_loss,cv_F1\n")

        summary_writer = tf.summary.FileWriter(
             log_dir, graph=tf.get_default_graph()
        )

        sess.run(tf.global_variables_initializer())
        tf.summary.scalar("train_loss", cross_entropy_loss)
        tf.summary.scalar("train_f1", f1)
        merged_summary_op = tf.summary.merge_all()

        global_iteration_idx = 0
        for i in range(epochs):
```

```
            print("Epoch %d" % i)
            ii = 0
            for batch_image,batch_label in get_batches_train(batch_size,
augmentation=True):
                ii += 1
                global_iteration_idx += 1
                train_op_,cross_entropy_loss_,summary_str,f1_,lr_ =\
             sess.run(
                         [train_op, cross_entropy_loss,
                   merged_summary_op, f1, lr],
                         feed_dict={
                            input_image: batch_image,
                            correct_label: batch_label,
                            learning_rate : g_lr,
                            keep_prob : 0.5
                })
                summary_writer.add_summary(
             summary_str,
             global_iteration_idx
             )
                print("Eproch %d, Iteration %d, loss = %1.5f, f1 = %1.5f, lr
= %1.5f" % (i, ii, cross_entropy_loss_, f1_, lr_))

            # Save the model every eproch
            l_f1_cv = []
            l_loss_cv = []

            for batch_image,batch_label in get_batches_cv(batch_size,
augmentation=False):
                cross_entropy_loss_, f1_ = sess.run(
                         [cross_entropy_loss_cv, f1_cv],
                         feed_dict={
                            input_image: batch_image,
                            correct_label: batch_label,
                            keep_prob : 0.5
                })
                l_loss_cv.append(cross_entropy_loss_)
                l_f1_cv.append(f1_)

            np_loss_cv = np.array(l_loss_cv)
            np_f1_cv   = np.array(l_f1_cv)
            f_out.write("%d,%1.5f,%1.5f\n" % (i, np.nanmean(np_loss_cv),
np.nanmean(np_f1_cv)))
```

```
            print("Validation, Eproch %d, loss = %1.5f, f1 = %1.5f" % (i,
np.nanmean(np_loss_cv), np.nanmean(np_f1_cv)))
            tf.train.write_graph(sess.graph_def, model_dir,
                                'eproch_%d_loss' % (i), as_text=False)
            saver.save(sess, '%s/eproch_%d_loss' % (model_dir, i))
```

至此，整个模型所需的所有模块定义完成了。下一步完成主函数，定义一些必要的变量，然后将所有模块串联起来：

```
    def main():
        image_shape = (256, 256)
        num_classes = 2
        random.seed(0)

        """
        获取所有 800 数据的样本名称
        以 8:2 比例划分所有样本的训练集以及验证集
        注意这里为了方便分析，已经将病理切片图片统一放置在 ./FCN/image/merge/ 文件夹中
        """
        l_sample = os.listdir("./FCN/image/merge/")
        l_sample = [ s.split(".tiff")[0] for s in l_sample]
        random.shuffle(l_sample)
        cv_ratio = 0.8
        split_idx = int(len(l_sample)*cv_ratio)
        l_sample_train = l_sample[0:split_idx]
        l_sample_cv = l_sample[split_idx:]

        get_batches_train = gen_batch_func(l_sample_train, image_shape)
        get_batches_cv = gen_batch_func(l_sample_cv, image_shape)

        """
        模型预计占用 10GB 左右显存。这里设置 tensorflow 不一次性耗尽显卡的所有显存
        """
        config = tf.ConfigProto()
        config.gpu_options.allow_growth = True

        with tf.Session(config=config) as sess:
            config = tf.ConfigProto()

            """
            模型训练准备。设置使用生成器，以及占位符
            """
            vgg_path = "./FCN/vgg/" # 下载 pb 格式的 VGG16 模型解压缩目录
            get_batches_fn = gen_batch_func(l_sample, image_shape)
            epochs = 30
```

```
        batch_size = g_batch_size
        correct_label = tf.placeholder(
         tf.int32, [None, None, None, num_classes]
    )
        learning_rate = tf.placeholder(tf.float32)

        """
        使用定义的函数，构建模型，规定优化目标，最后进行模型的训练
        """
        input_image, keep_prob, vgg_layer3_out, vgg_layer4_out, \
          vgg_layer7_out = load_vgg(sess, vgg_path)

        nn_last_layer = layers(vgg_layer3_out,
                      vgg_layer4_out,
                      vgg_layer7_out,
                      num_classes
         )

        pred_label, training_op, cross_entropy_loss, f1, lr = \
            optimize(nn_last_layer, correct_label, learning_rate,
               num_classes, batch_size, split_idx
         )
        cross_entropy_loss_cv, f1_cv = \
            validation(nn_last_layer, correct_label, num_classes)

        train_nn(sess, epochs, batch_size, get_batches_train,
          get_batches_cv, training_op, cross_entropy_loss,
                cross_entropy_loss_cv, f1, f1_cv, lr, input_image,
               correct_label, keep_prob, learning_rate)
```

11.4 程序执行

最后在 runfcn.py 中，将学习率、正则化参数以及批次数据大小作为 argv 环境参数，这样就可以通过 Linux 脚本来自动寻找最优参数组合：

展示脚本的内容：

```
!head runfcn.sh
```

结果如下：

```
python runfcn.py 1e-3 1e-2 2
python runfcn.py 1e-4 1e-2 2
python runfcn.py 1e-5 1e-2 2
```

```
python runfcn.py 1e-3 1e-2 4
python runfcn.py 1e-4 1e-2 4
python runfcn.py 1e-5 1e-2 4
python runfcn.py 1e-3 1e-2 8
python runfcn.py 1e-4 1e-2 8
python runfcn.py 1e-5 1e-2 8
python runfcn.py 1e-3 1e-6 2
```

其中有希望的各种参数组合，可以通过直接执行脚本，训练模型，继而将不同参数训练的模型分别保存下来：

```
!sh runfcn.sh
```

11.5 模型结果可视化

11.5.1 加载函数

加载需要用到的函数：

```
import re
import os
import cv2
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.utils import shuffle
from cairosvg import svg2png
import matplotlib.pyplot as plt
import random
from tqdm import tqdm
import tensorflow as tf
from keras.preprocessing.image import transform_matrix_offset_center
import scipy
import scipy.ndimage as ndi

from runfcn import *

%matplotlib inline
```

11.5.2 选择验证集并编写预测函数

训练过程中，我们已经按照 8:2 的比例定义了训练集以及验证集，这里进行可视化时仍然使用这个定义，主要看验证集中的结果如何，进而写出预测函数。

```
image_shape = (256, 256)
num_classes = 2
random.seed(0)

l_sample = !ls ./FCN/image/merge/*tiff
l_sample = [ s.split("/")[-1].split(".tiff")[0] for s in l_sample]
random.shuffle(l_sample)

cv_ratio = 0.8
split_idx = int(len(l_sample)*cv_ratio)
get_batches_train = gen_batch_func(l_sample[0:split_idx], image_shape)
get_batches_cv    = gen_batch_func(l_sample[split_idx:],  image_shape)

def predict_using_raw_model(l_image_file, model_file, batch_size, split_idx):
    vgg_path = "./FCN/vgg/"
    num_classes = 2
    tf.reset_default_graph()
    with tf.Session() as sess:
        correct_label = tf.placeholder(tf.int32,
         [None, None, None, num_classes]
   )
        learning_rate = tf.placeholder(tf.float32)

        input_image, keep_prob, vgg_layer3_out, vgg_layer4_out,\
         vgg_layer7_out = load_vgg(sess, vgg_path)

        nn_last_layer = layers(vgg_layer3_out, vgg_layer4_out,
         vgg_layer7_out, num_classes
   )
        logits, training_op, cross_entropy_loss, f1, lr = \
            optimize(nn_last_layer, correct_label, learning_rate,
                num_classes, batch_size, split_idx
         )

        saver = tf.train.Saver()
        saver.restore(sess, model_file)
        l_street_im = []
        for image_file in tqdm(l_image_file):
            result_file = "%s.region.png" % (image_file.split(".tiff")[0])
            image_raw = scipy.misc.imread(image_file)
            image_raw_shape = image_raw.shape[0:2]
            image = scipy.misc.imresize(image_raw, image_shape)
            im_softmax = sess.run(
                [tf.nn.softmax(logits)],
                {keep_prob: 1.0, input_image: [image]}
```

```
            )

              im_softmax = im_softmax[0][:, 1].reshape(
                  image_shape[0], image_shape[1]
            )
              segmentation = (im_softmax < 0.5).reshape(
            image_shape[0], image_shape[1], 1
            )

              mask = np.dot(segmentation,
                  np.array([[0, 255, 0, 127]])
            )
              mask = scipy.misc.imresize(mask, image_raw_shape)
              mask = scipy.misc.toimage(mask, mode="RGBA")
              street_im = scipy.misc.toimage(image_raw)
              street_im.paste(mask, box=None, mask=mask)
              l_street_im.append(street_im)

              gray = np.array((im_softmax < 0.5)*255, dtype=np.uint8)
              gray = cv2.resize(gray, image_raw_shape)
              cv2.imwrite(result_file, gray)

        return l_street_im
```

11.5.3 根据 tensorborad 可视化结果选择最好的模型

上文写道，在 runfcn.py 中将学习率、正则化参数以及批次数据大小作为 argv 环境参数。通过这种方式，可以训练多组参数，从中寻找最优的一组参数，这组参数对应的模型，可以认为就是最优的模型。

那么如何找出其中最好的一个模型呢？大家可以使用 tensorborad 对结果进行可视化。tensorborad 在环境中的使用方式如下：

- 回到 http://localhost:8888/，单击右上角的 new，添加 Terminal。
- 在打开的黑色终端中，输入：

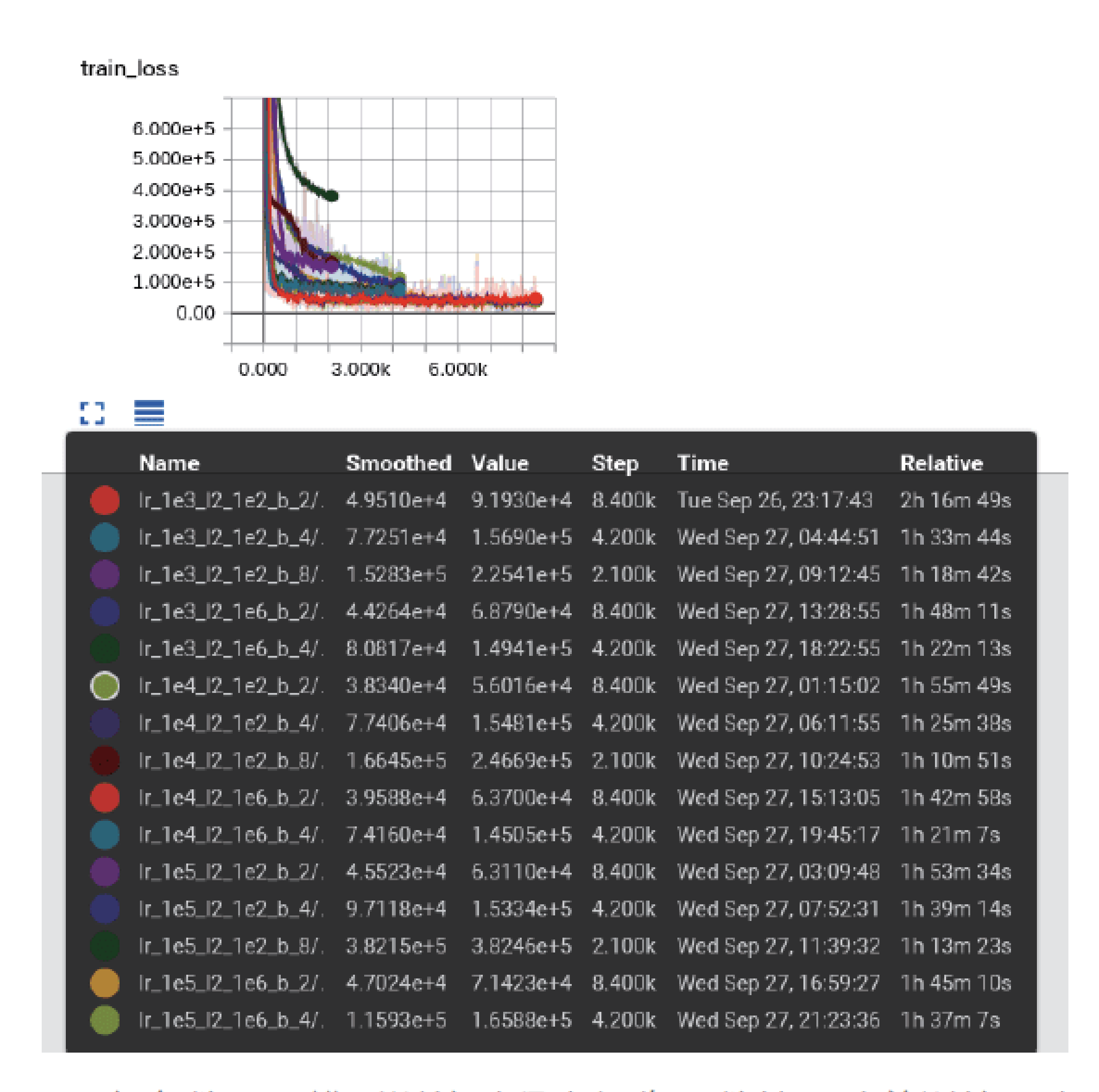
```
    # 使用方法：
    tensorboard --logdir [名称1]:[tensorborad文件夹1],[名称2]:[tensorborad文件夹2]...

    # 对于我们这里跑出来的模型：
    tensorboard --logdir lr_1e3_l2_1e2_b_2:usingNonAug_logs_l2_norm_lr_1.00e-03 _l2_1.00e-02_e10_batch_2,…,lr_1e5_l2_1e6_b_4:usingNonAug_logs_l2_norm_lr_1.00e-05_l2_1.00e-06_e10_batch_4
```

- 打开 http://localhost:8888/，等待模型完成加载。

结果如下：

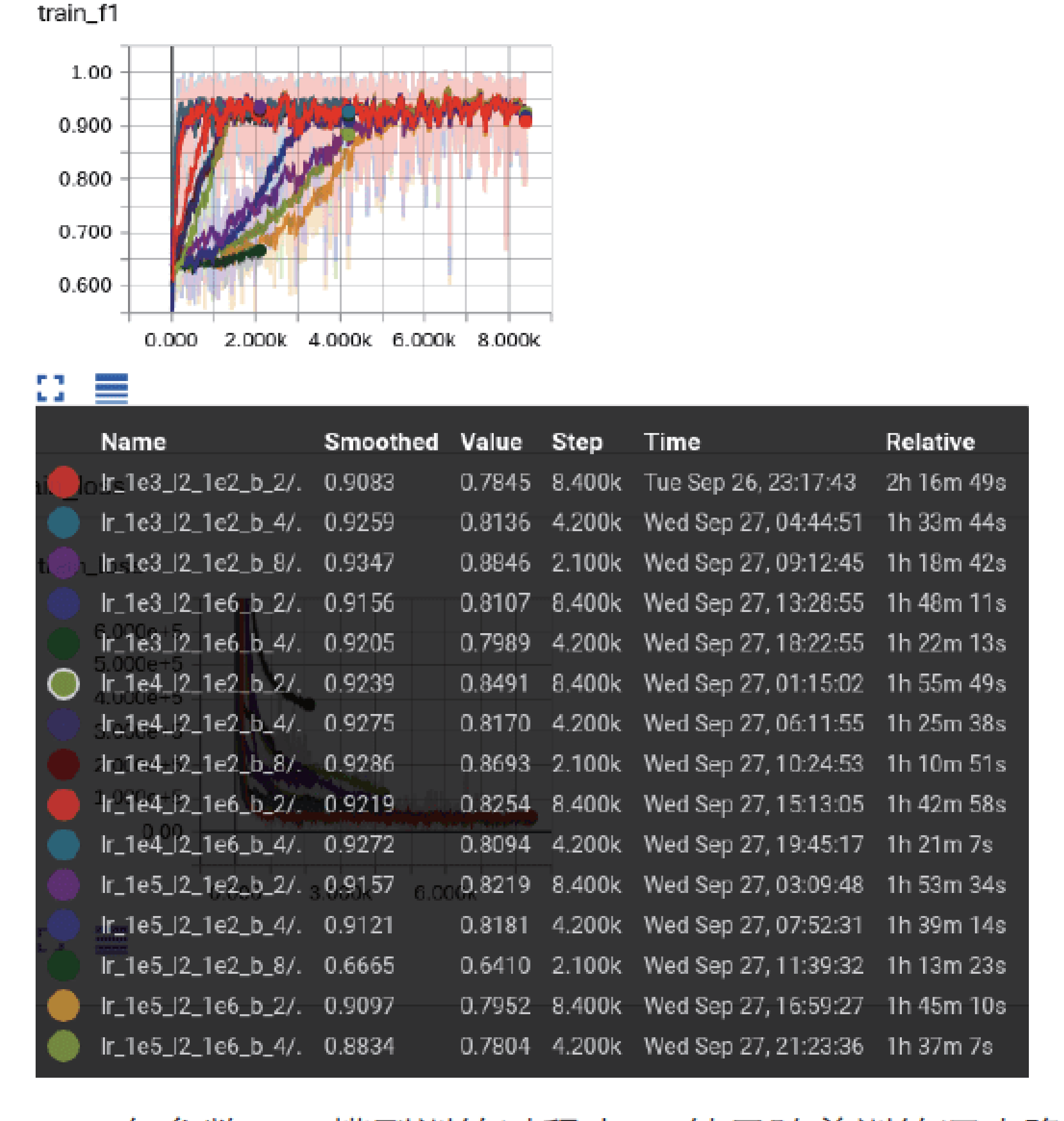
首先看训练损失函数 loss 的结果，如图 11-4 所示。我们发现 learningRate=1e-4、正则化常数 l2=1e-2、批数据大小 batch_size=2 的情况下，模型在训练数据中的损失最小。



Name	Smoothed	Value	Step	Time	Relative
lr_1e3_l2_1e2_b_2/.	4.9510e+4	9.1930e+4	8.400k	Tue Sep 26, 23:17:43	2h 16m 49s
lr_1e3_l2_1e2_b_4/.	7.7251e+4	1.5690e+5	4.200k	Wed Sep 27, 04:44:51	1h 33m 44s
lr_1e3_l2_1e2_b_8/.	1.5283e+5	2.2541e+5	2.100k	Wed Sep 27, 09:12:45	1h 18m 42s
lr_1e3_l2_1e6_b_2/.	4.4264e+4	6.8790e+4	8.400k	Wed Sep 27, 13:28:55	1h 48m 11s
lr_1e3_l2_1e6_b_4/.	8.0817e+4	1.4941e+5	4.200k	Wed Sep 27, 18:22:55	1h 22m 13s
lr_1e4_l2_1e2_b_2/.	3.8340e+4	5.6016e+4	8.400k	Wed Sep 27, 01:15:02	1h 55m 49s
lr_1e4_l2_1e2_b_4/.	7.7406e+4	1.5481e+5	4.200k	Wed Sep 27, 06:11:55	1h 25m 38s
lr_1e4_l2_1e2_b_8/.	1.6645e+5	2.4669e+5	2.100k	Wed Sep 27, 10:24:53	1h 10m 51s
lr_1e4_l2_1e6_b_2/.	3.9588e+4	6.3700e+4	8.400k	Wed Sep 27, 15:13:05	1h 42m 58s
lr_1e4_l2_1e6_b_4/.	7.4160e+4	1.4505e+5	4.200k	Wed Sep 27, 19:45:17	1h 21m 7s
lr_1e5_l2_1e2_b_2/.	4.5523e+4	6.3110e+4	8.400k	Wed Sep 27, 03:09:48	1h 53m 34s
lr_1e5_l2_1e2_b_4/.	9.7118e+4	1.5334e+5	4.200k	Wed Sep 27, 07:52:31	1h 39m 14s
lr_1e5_l2_1e2_b_8/.	3.8215e+5	3.8246e+5	2.100k	Wed Sep 27, 11:39:32	1h 13m 23s
lr_1e5_l2_1e6_b_2/.	4.7024e+4	7.1423e+4	8.400k	Wed Sep 27, 16:59:27	1h 45m 10s
lr_1e5_l2_1e6_b_4/.	1.1593e+5	1.6588e+5	4.200k	Wed Sep 27, 21:23:36	1h 37m 7s

图 11-4　各参数下，模型训练过程中损失函数结果随着训练逐步降低

同样，在 learningRate=1e-4、正则化常数 l2=1e-2、批数据大小 batch_size=2 的情况下，模型在训练数据中 f1 得分最高，如图 11-5 所示。



Name	Smoothed	Value	Step	Time	Relative
lr_1e3_l2_1e2_b_2/.	0.9083	0.7845	8.400k	Tue Sep 26, 23:17:43	2h 16m 49s
lr_1e3_l2_1e2_b_4/.	0.9259	0.8136	4.200k	Wed Sep 27, 04:44:51	1h 33m 44s
lr_1e3_l2_1e2_b_8/.	0.9347	0.8846	2.100k	Wed Sep 27, 09:12:45	1h 18m 42s
lr_1e3_l2_1e6_b_2/.	0.9156	0.8107	8.400k	Wed Sep 27, 13:28:55	1h 48m 11s
lr_1e3_l2_1e6_b_4/.	0.9205	0.7989	4.200k	Wed Sep 27, 18:22:55	1h 22m 13s
lr_1e4_l2_1e2_b_2/.	0.9239	0.8491	8.400k	Wed Sep 27, 01:15:02	1h 55m 49s
lr_1e4_l2_1e2_b_4/.	0.9275	0.8170	4.200k	Wed Sep 27, 06:11:55	1h 25m 38s
lr_1e4_l2_1e2_b_8/.	0.9286	0.8693	2.100k	Wed Sep 27, 10:24:53	1h 10m 51s
lr_1e4_l2_1e6_b_2/.	0.9219	0.8254	8.400k	Wed Sep 27, 15:13:05	1h 42m 58s
lr_1e4_l2_1e6_b_4/.	0.9272	0.8094	4.200k	Wed Sep 27, 19:45:17	1h 21m 7s
lr_1e5_l2_1e2_b_2/.	0.9157	0.8219	8.400k	Wed Sep 27, 03:09:48	1h 53m 34s
lr_1e5_l2_1e2_b_4/.	0.9121	0.8181	4.200k	Wed Sep 27, 07:52:31	1h 39m 14s
lr_1e5_l2_1e2_b_8/.	0.6665	0.6410	2.100k	Wed Sep 27, 11:39:32	1h 13m 23s
lr_1e5_l2_1e6_b_2/.	0.9097	0.7952	8.400k	Wed Sep 27, 16:59:27	1h 45m 10s
lr_1e5_l2_1e6_b_4/.	0.8834	0.7804	4.200k	Wed Sep 27, 21:23:36	1h 37m 7s

图 11-5　各参数下，模型训练过程中 f1 结果随着训练逐步降低

最后在验证集中确认结果，同样发现这一组参数表现最好：

```
%%bash
for i in 'ls usingNonAug_cv_lr_1.00e-0*csv'
do echo $i && tail -n 1 $i
done |  paste - - | grep -v F1 | sed 's/,/\t/g' | sort -nk 3
usingNonAug_cv_lr_1.00e-04_l2_1.00e-02_e10_batch_2.csv 29 33924.51562
0.93591 usingNonAug_cv_lr_1.00e-04_l2_1.00e-06_e10_batch_2.csv 29
34027.92578 0.93794 usingNonAug_cv_lr_1.00e-03_l2_1.00e-06_e10_batch_2.csv
29 37747.60547 0.93466 usingNonAug_cv_lr_1.00e-05_l2_1.00e-02_e10_batch_2.
csv 29 40629.83594 0.93062 usingNonAug_cv_lr_1.00e-05_l2_1.00e-06_e10_
batch_2.csv 29 41357.10547 0.93023 usingNonAug_cv_lr_1.00e-03_l2_1.00e-02_
e10_batch_2.csv 29 44756.12891 0.93455 usingNonAug_cv_lr_1.00e-03_
l2_1.00e-02_e10_batch_4.csv 29 70589.04688 0.93778 usingNonAug_cv_
lr_1.00e-04_l2_1.00e-02_e10_batch_4.csv 29 72766.92969 0.93526 usingNonAug_
cv_lr_1.00e-04_l2_1.00e-06_e10_batch_4.csv 29 75052.12500 0.93443
usingNonAug_cv_lr_1.00e-03_l2_1.00e-06_e10_batch_4.csv 29 78452.96094
0.93639 usingNonAug_cv_lr_1.00e-05_l2_1.00e-02_e10_batch_4.csv 29
97774.84375 0.91497 usingNonAug_cv_lr_1.00e-05_l2_1.00e-06_e10_batch_4.csv
29 115415.34375 0.88782 usingNonAug_cv_lr_1.00e-03_l2_1.00e-02_e10_batch_8.
csv 29 149054.15625 0.93808 usingNonAug_cv_lr_1.00e-04_l2_1.00e-02_e10_
batch_8.csv 29 156687.70312 0.93564 usingNonAug_cv_lr_1.00e-05_l2_1.00e-02_
e10_batch_8.csv 29 374531.71875 0.66432
```

11.5.4 尝试逐步降低学习率

对于之前的结果，直接用 Adam(1e-4) 的参数进行模型训练。我们考虑将学习率逐步降低，会不会表现得更好。注意，这里 Adam 属于自适应学习率的代表，根据第 7 章的算法、代码来看，就是尽管这里用了 1e-4 的学习率，实际优化过程中，这个学习率是不断迭代减小的，设置 learning_rate decay 的必要性并不高。这里保留此部分内容，读者可以在尝试基于动量的优化算法的尝试过程中引入这种策略。

实际表现也符合预期，加入一个指数衰减的学习率下降操作以后，并没有很好地提升：

```
%%bash
tail usingNonAug_cv_ExpDecay_lr_1.00e-04_l2_1.00e-02_e10_batch_2.csv \
    usingNonAug_cv_lr_1.00e-04_l2_1.00e-02_e10_batch_2.csv
==> usingNonAug_cv_ExpDecay_lr_1.00e-04_l2_1.00e-02_e10_batch_2.csv <==
20,36099.92188,0.93504
21,35943.85156,0.93427
22,39463.47266,0.92656
23,36184.65625,0.93661
24,35735.08594,0.93603
```

```
25,35674.34766,0.93512
26,40937.39062,0.91563
27,37334.96094,0.93343
28,35837.04688,0.93899
29,35439.44141,0.93295

==> usingNonAug_cv_lr_1.00e-04_l2_1.00e-02_e10_batch_2.csv <==
20,35261.18359,0.93672
21,34241.35547,0.93779
22,41248.08203,0.93692
23,36137.74609,0.93574
24,35158.97656,0.93502
25,35176.26562,0.93528
26,39708.64844,0.91887
27,35172.44531,0.93328
28,36062.39062,0.94078
29,33924.51562,0.93591
```

11.6 观察模型在验证集上的预测表现

将这一组参数训练的模型运用在验证集上，看看结果如何：

```
l_image_file = [
   "./FCN/image/merge/%s.tiff" % x for x in l_sample[split_idx:]
]
l_label_file = [
   "./FCN/labels/%s.png" % x for x in l_sample[split_idx:]
]

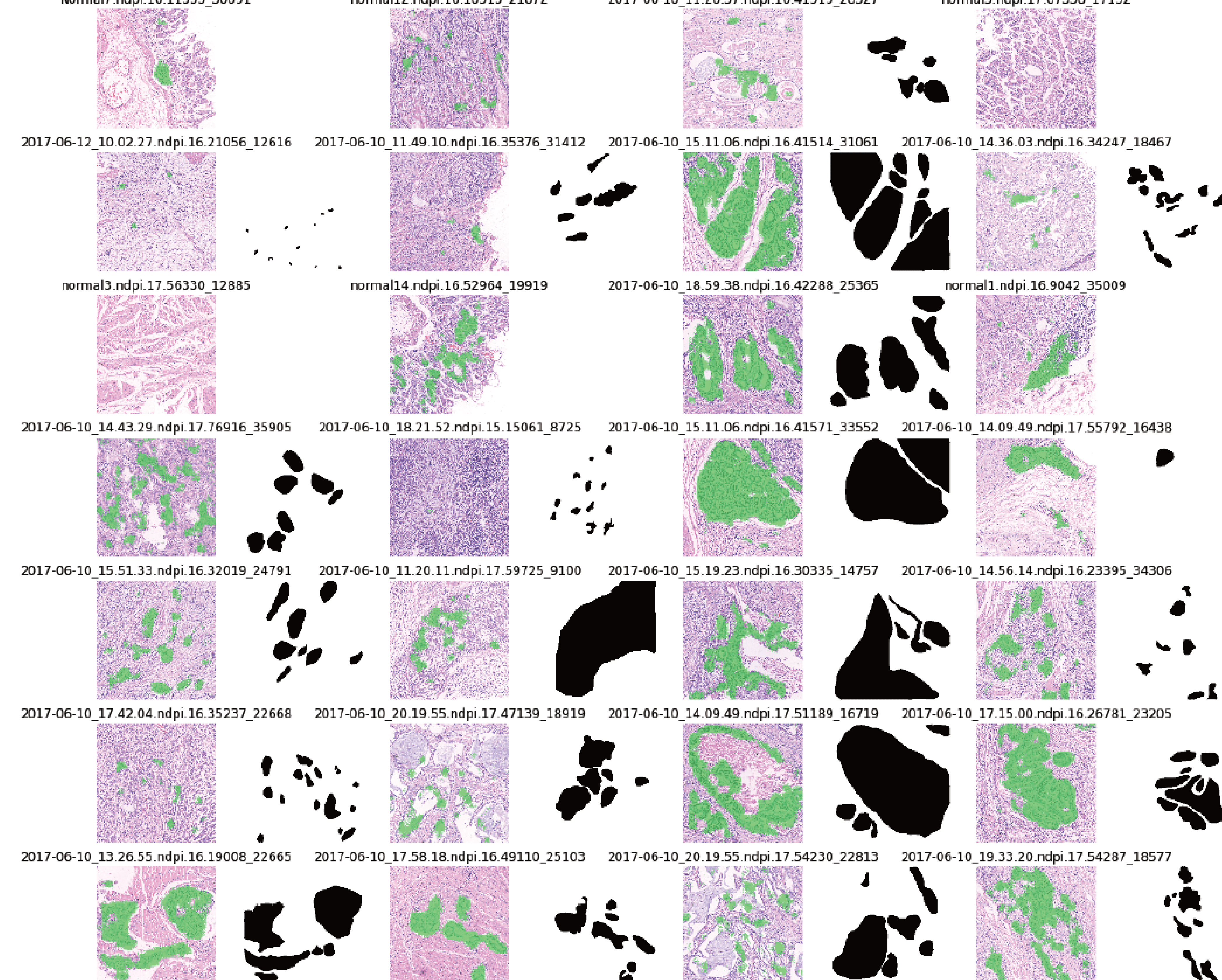
select_model = "./usingNonAug_models_l2_norm_ExpDecay_lr_1.00e-04_l2_
1.00e-02_e10_batch_2/eproch_29_loss"
l_im = predict_using_raw_model(
   l_image_file,
          model_file=select_model,
          batch_size=g_batch_size,
   split_idx=split_idx
)
```

对验证集前 64 个样本进行预测。这些图片一行 4 个样本，每个样本两张图片，其中左边绿色部分是模型预测的癌症区域，右边黑色区域是真实癌症区域的标注。

```
fig = plt.figure(figsize=(20,20))
```

```
    for idx in range(32):
        ax1 = fig.add_subplot(8,8,2*idx+1)
        ax2 = fig.add_subplot(8,8,2*idx+2)
        img_predict = l_im[idx]
        try:
            img_groundtruth = scipy.misc.imread(l_label_file[idx])
        except:
            img_groundtruth = np.zeros_like(l_im[idx])+255

        ax1.imshow(img_predict)
        ax2.imshow(img_groundtruth[:,:,0], "Greys")
        ax1.set_axis_off()
        ax2.set_axis_off()
        ax1.set_title(l_sample[split_idx:][idx].split(".2048")[0])
```



```
fig = plt.figure(figsize=(20,20))

for idx in range(32):
    ax1 = fig.add_subplot(8,8,2*idx+1)
    ax2 = fig.add_subplot(8,8,2*idx+2)
    idx += 32
    img_predict = l_im[idx]
    try:
        img_groundtruth = scipy.misc.imread(l_label_file[idx])
    except:
        img_groundtruth = np.zeros_like(l_im[idx])+255

    ax1.imshow(img_predict)
    ax2.imshow(img_groundtruth[:,:,0], "Greys")
    ax1.set_axis_off()
    ax2.set_axis_off()
    ax1.set_title(l_sample[split_idx:][idx].split("2048")[0])
```

2017-06-10_15.56.34.ndpi.16.27855_20636. Normal5.ndpi.16.18723_34019. 2017-06-10_18.47.26.ndpi.17.48562_18196. 2017-06-10_15.11.06.ndpi.16.36799_30479.

2017-06-10_15.19.23.ndpi.16.22382_26073. 2017-06-12_09.57.54.ndpi.16.14920_9644. 2017-06-10_15.56.34.ndpi.16.30614_21285. 2017-06-12_09.57.54.ndpi.16.20278_13750.

2017-06-10_18.34.21.ndpi.17.35599_31081. normal1.ndpi.16.5702_35104. normal12.ndpi.16.6461_15577. 2017-06-10_11.45.05.ndpi.16.44030_24562.

2017-06-10_14.38.49.ndpi.17.42318_24319. 2017-06-10_14.36.03.ndpi.16.20304_13766. 2017-06-10_13.30.55.ndpi.17.42157_34785. 2017-06-10_13.03.44.ndpi.16.31950_7629.

2017-06-10_20.15.16.ndpi.17.57414_21031. normal2.ndpi.17.48408_4201. 2017-06-10_11.45.05.ndpi.16.29143_17782. 2017-06-10_14.56.14.ndpi.16.22562_27255.

2017-06-10_14.56.14.ndpi.16.23392_37214. 2017-06-10_14.49.45.ndpi.16.39465_15515. 2017-06-10_16.17.59.ndpi.16.25066_18339. 2017-06-10_15.19.23.ndpi.16.28681_29311.

Normal4.ndpi.15.8691_10949. 2017-06-10_16.54.09.ndpi.16.46723_27991. normal9.ndpi.17.57584_12532. Normal7.ndpi.16.10235_32678.

normal12.ndpi.16.7043_24176. 2017-06-10_19.33.20.ndpi.17.36408_14798. 2017-06-10_16.17.59.ndpi.16.18597_20492. Normal6.ndpi.17.38558_18379.

由此可见，这里除了颜色偏深紫色且细胞排列较致密的样本出现明显漏检以外，其他情况下主要还是假阳性。这样，就可以把模型当作一个实习医生，对病理切片做一个初筛，然后进一步将结果给专家医师确认。

11.7 参考文献及网页链接

［1］ Data Dream 病理切片识别 AI 比赛 . Available at: http://www.datadreams.org/race-data-3.html.

［2］ Shelhamer, E., Long, J. & Darrell, T. Fully Convolutional Networks for Semantic Segmentation. [1605.06211] Fully Convolutional Networks for Semantic Segmentation (2016). Available at: https://arxiv.org/abs/1605.06211.

［3］ Tai, L., Ye, H., Ye, Q. & Liu, M. PCA-aided Fully Convolutional Networks for Semantic Segmentation of Multi-channel fMRI. [1610.01732] PCA-aided Fully Convolutional Networks for Semantic Segmentation of Multi-channel fMRI (2017). Available at: https://arxiv.org/abs/1610.01732.

第 12 章

知行合一——如何写一个深度学习 App

本章将制作一个可以在 iPhone 上利用摄像头识别猫狗，并且可视化卷积神经网络关注的区域的一个应用程序。

12.1 CAM 可视化

首先回顾一下在 5.3.3 节讲过的全局平均池化层（GlobalAveragePooling，GAP）的相关知识点。GAP 通常用在卷积神经网络的最后一层，用于降维，使用 GAP 层可以极大地降低特征的维度，使全连接层参数不至于过多，保留了特征的强度信息，丢弃了特征的位置信息，因为是分类问题，对位置信息不敏感，所以使用 GAP 层来降维效果很好。

周博磊对 GAP 层进行了思考后指出，可以对卷积层的输出加权平均，权重是 GAP 层到这个分类的权重，如图 12-1 所示。

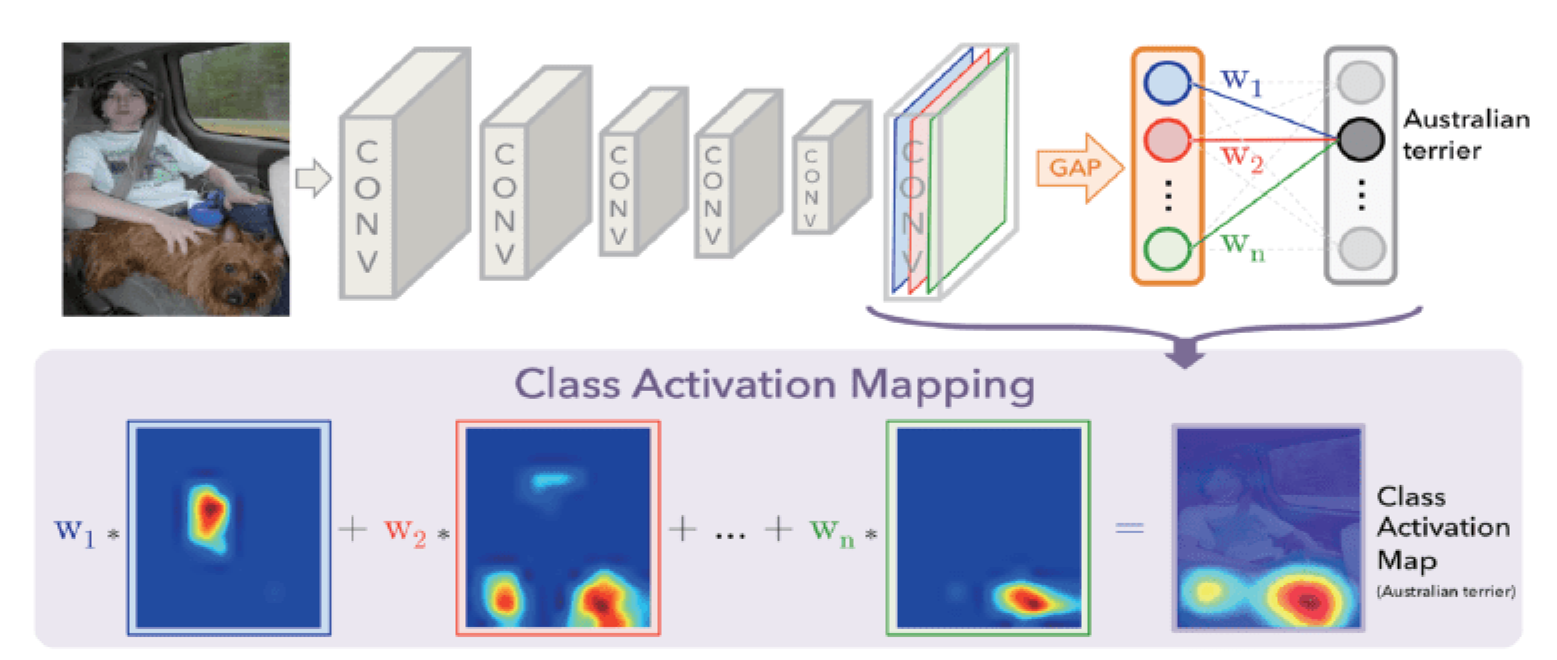


Figure 2. Class Activation Mapping: the predicted class score is mapped back to the previous convolutional layer to generate the class activation maps (CAMs). The CAM highlights the class-specific discriminative regions.

图 12-1 CAM 可视化的原理

这样就能得到一个 CAM（Class Activation Mapping）可视化图，也就是类激活图。简单来说，可以在最后一层卷积层后面加一层。

参考链接：http://cnnlocalization.csail.mit.edu/

12.2 导出分类模型和 CAM 可视化模型

本节会介绍如何训练一个识别猫狗的模型，导出对应的分类器以及输出 CAM 可视化图的模型。

12.2.1 载入数据集

载入数据集的代码如下：

```
    import cv2
    import numpy as np
    from tqdm import tqdm

    n = 25000
    width = 224

    X = np.zeros((n, width, width, 3), dtype=np.uint8)
    y = np.zeros((n, 2), dtype=np.uint8)

    for i in tqdm(range(n/2)):
        X[i] = cv2.resize(cv2.imread('train/cat.%d.jpg' % i), (width,
width))
        X[i+n/2] = cv2.resize(cv2.imread('train/dog.%d.jpg' % i), (width,
width))

    y[:n/2, 0] = 1
    y[n/2:, 1] = 1
```

这里与之前的猫狗大战不太一样的地方就是 y 变成了两个维度，因为需要把猫编码成 [1, 0]、狗编码成 [0, 1]，才能针对分类进行 CAM 可视化。

如果是一个神经元 sigmoid 做二分类，模型只会把狗对应的权值加大，对模型来说，猫和其他背景没有差别，因此需要弄两个维度，然后用 softmax 激活函数，这样模型就不能仅分辨狗和非狗了，因为如果把猫也看作背景，softmax 之后没办法让猫的输出比狗大。

12.2.2 提取特征

提取特征的代码如下：

```
from keras.layers import *
from keras.models import *
from keras.applications import *
from keras.optimizers import *
from keras.regularizers import *

def preprocess_input(x):
    return x - [103.939, 116.779, 123.68]


def get_features(MODEL, data=X):
    cnn_model = MODEL(include_top=False, input_shape=(width, width, 3), 
weights='imagenet')

    inputs = Input((width, width, 3))
    x = inputs
    x = Lambda(preprocess_input, name='preprocessing')(x)
    x = cnn_model(x)
    x = GlobalAveragePooling2D()(x)
    cnn_model = Model(inputs, x)

    features = cnn_model.predict(data, batch_size=64, verbose=1)
    return features

features = get_features(ResNet50, X)
```

这里使用经过 ImageNet 预训练的 ResNet50 提取特征。

12.2.3 搭建和训练分类器

搭建和训练分类器的代码如下：

```
inputs = Input(features.shape[1:])
x = inputs
x = Dropout(0.5)(x)
```

```
    x = Dense(2, activation='softmax', kernel_regularizer=l2(1e-4), bias_
regularizer=l2(1e-4))(x)
    model = Model(inputs, x, name='prediction')
    model.compile(optimizer='adam',
                  loss='binary_crossentropy',
                  metrics=['accuracy'])
    h = model.fit(features, y, batch_size=128, epochs=10, validation_split=0.2)
    # Epoch 10/10 loss: 0.0336 - acc: 0.9889 - val_loss: 0.0453 - val_acc: 0.9856
```

这个模型非常简单，直接搭建一个全连接分类器，然后用 adam 优化器来训练就好了。模型在第 10 代的时候，准确率可以达到 98.5% 以上。

12.2.4 搭建分类模型和 CAM 模型

首先获取刚才训练的模型的全连接权值：

```
    weights = model.get_weights()[0]
```

然后搭建模型时，要注意这一点，目前 Keras 官方提供的 ResNet50 最后一层带有一个（7, 7）的池化层，但是需要输出卷积层的原始数据，而不是把每个激活图压缩成一个点，因此取倒数第二层搭建成一个新的 cnn_model，再去搭建 CAM 模型。

在对卷积层输出的激活图进行加权平均的时候，可以理解为是卷积核大小为 1×1 的不带 bias 的卷积层。

分类器就简单了，直接用 GlobalAveragePooling2D 进行平均，然后用刚才训练的 model 算一下就好了。

```
    cnn_model = ResNet50(include_top=False,
                         input_shape=(width, width, 3),
                         weights='imagenet'
    )
    cnn_model = Model(cnn_model.input, cnn_model.layers[-2].output,
name='resnet50')

    inputs = Input((width, width, 3))
    x = inputs
    x = cnn_model(x)
    cam = Conv2D(2, 1, use_bias=False, name='cam')(x)
    model_cam = Model(inputs, cam)

    x = GlobalAveragePooling2D(name='gap')(x)
    x = model(x)
    model_clf = Model(inputs, x)
```

载入权值的时候需要把 weights 从（2048, 2）reshape 成（1, 1, 2048, 2），然后载入到 model_cam 的最后一个 1×1 卷积层里。

```
model_cam.layers[-1].set_weights([weights.reshape((1, 1, 2048, 2))])
```

搭建好以后模型的可视化如图 12-2 所示。

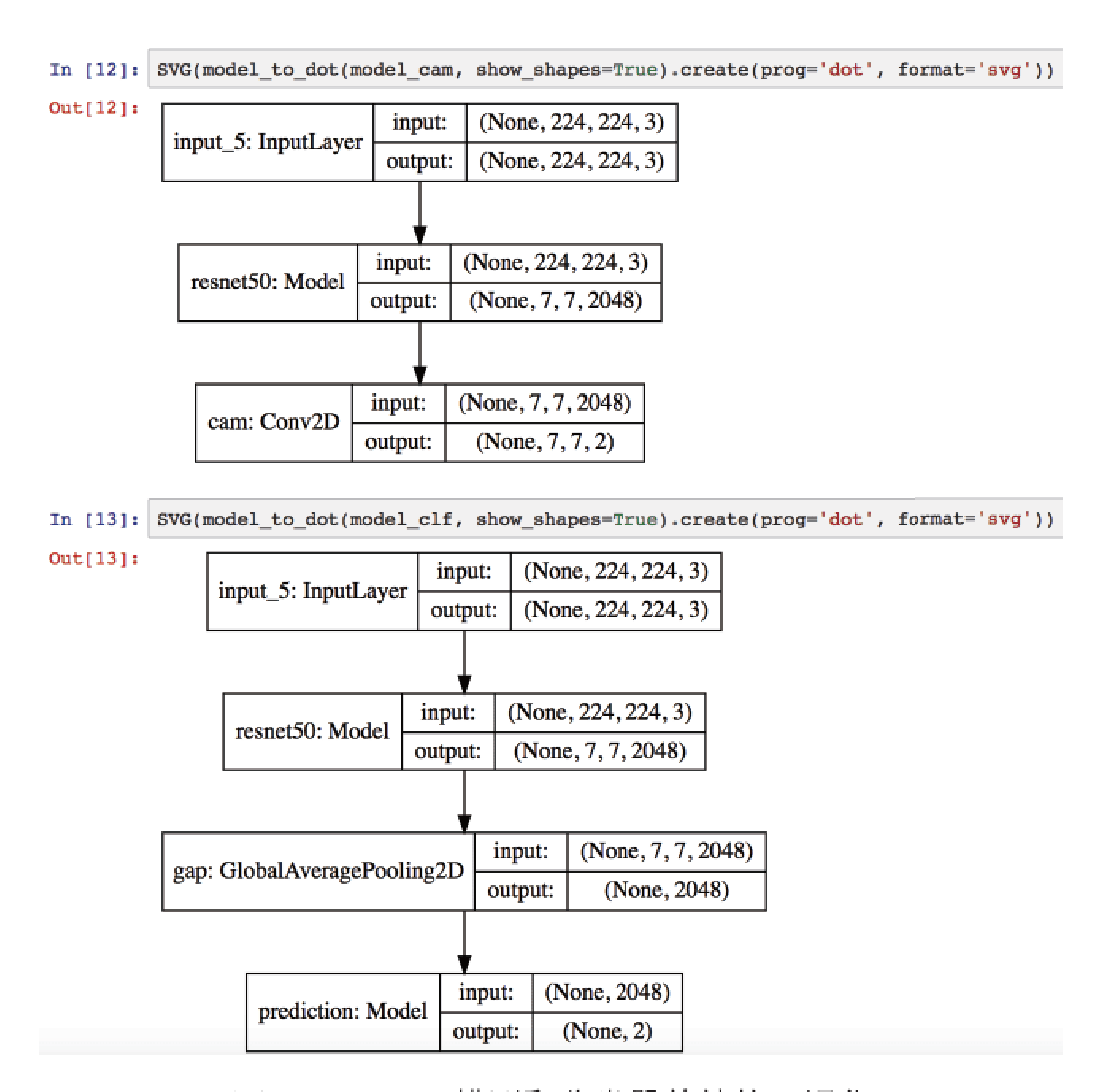


图 12-2　CAM 模型和分类器的结构可视化

12.2.5　可视化测试

我们可以利用刚才搭建的两个模型尝试可视化，首先用 model_clf 预测这张图片是猫还是狗。它会输出两个概率：第一个是猫的，第二个是狗的，我们取猫的概率，也就是 prediction[0, 0]。

然后用 model_cam 输出两张 CAM 可视化的图，模型输出的 shape 是 (1, 7, 7, 2)，可以简单地用 cam[0, :, :, 1 if prediction < 0.5 else 0] 来提取对应类别的 CAM 图。

之后我们进行一些调整，首先将图片整体缩小 1/10，因为 CAM 图的数值范围大概在 -5~30，平均值大约是 6，所以经过几次调整，除以 10 是可视化效果比较好的数值，然后将数值限制在 0~1 之间，并转换为 Uint8（因为接下来的染色需要 Uint8 的格式）。

对 CAM 的染色使用 OpenCV 的函数以及颜色条（见图 12-13），我们选择了 COLORMAP_JET 的样式。



图 12-3　JET 颜色条

参考链接：http://docs.opencv.org/trunk/d3/d50/group__imgproc__colormap.html。

最后将染色的 heatmap 加在原图上，形成最终的可视化效果图（见图 12-4）。

```
import matplotlib.pyplot as plt
import random
%matplotlib inline
%config InlineBackend.figure_format = 'retina'

# 用模型进行预测
index = 13734
img = X[index]
prediction = model_clf.predict(np.expand_dims(img, 0))
prediction = prediction[0, 0]

cam = model_cam.predict(np.expand_dims(img, 0))
cam = cam[0, :, :, 1 if prediction < 0.5 else 0]

# 调整 CAM 的范围
cam /= 10
cam[cam < 0] = 0
cam[cam > 1] = 1
cam = cv2.resize(cam, (224, 224))
cam = np.uint8(255*cam)

# 染成彩色
heatmap = cv2.applyColorMap(cam, cv2.COLORMAP_JET)

# 加在原图上
out = cv2.addWeighted(img, 0.8, heatmap, 0.4, 0)

# 显示图片
plt.axis('off')
plt.imshow(out[:,:,::-1])
```



图 12-4 叠加了 CAM 可视化的图片

12.2.6 保存模型

保存模型的代码如下：

```
model_clf.save('model_clf.h5')
model_cam.save('model_cam.h5')
```

12.2.7 导出 mlmodel 模型文件

在 iOS 11 中，可以直接使用 Keras 的模型，只需要使用苹果的模型转换库 Core ML Tools 将 Keras 以 HDF5 格式存储的模型转换为苹果使用的 mlmodel 格式即可。

参考链接：https://developer.apple.com/documentation/coreml/converting_trained_models_to_core_ml。

这里会设置一些必要的参数，比如 RGB 的偏移，设置输入 / 输出的名字、介绍等，然后保存模型。

```
from coremltools.converters.keras import convert

coreml_model = convert('model_clf.h5',
                blue_bias=103.939,
                green_bias=116.779,
                red_bias=123.68,
                input_names=['image'],
                image_input_names='image',
                output_names='prediction'
)

coreml_model.author = 'YPW'
coreml_model.short_description = 'Dogs vs Cats'
coreml_model.license = 'MIT'
coreml_model.input_description['image'] = 'A 224x224 Image.'
coreml_model.output_description['prediction'] = 'The probability of Dog
and Cat.'
coreml_model.save('model_clf.mlmodel')
然后是 CAM 模型：
coreml_model = convert('model_cam.h5', blue_bias=103.939, green_bias=116.779,
                            red_bias=123.68, input_names=['image'],
                            image_input_names='image', output_
names='cam')

coreml_model.author = 'YPW'
coreml_model.short_description = 'Dogs vs Cats'
coreml_model.license = 'MIT'
coreml_model.input_description['image'] = 'A 224x224 Image.'
```

```
coreml_model.output_description['cam'] = 'The cam Image.'
coreml_model.save('model_cam.mlmodel')
```

12.3 开始编写 App

本节将开始编写一个 App，创建一个可以使用 OpenCV 调用摄像头读取图片的 App，这也是下一节使用深度学习模型的前提。首先创建工程，然后测试工程，最后配置工程。

12.3.1 创建工程

创建工程的操作步骤如下：

步骤 01 打开 Xcode 9，选择 Create a new Xcode project 选项，创建一个工程，如图 12-5 所示。

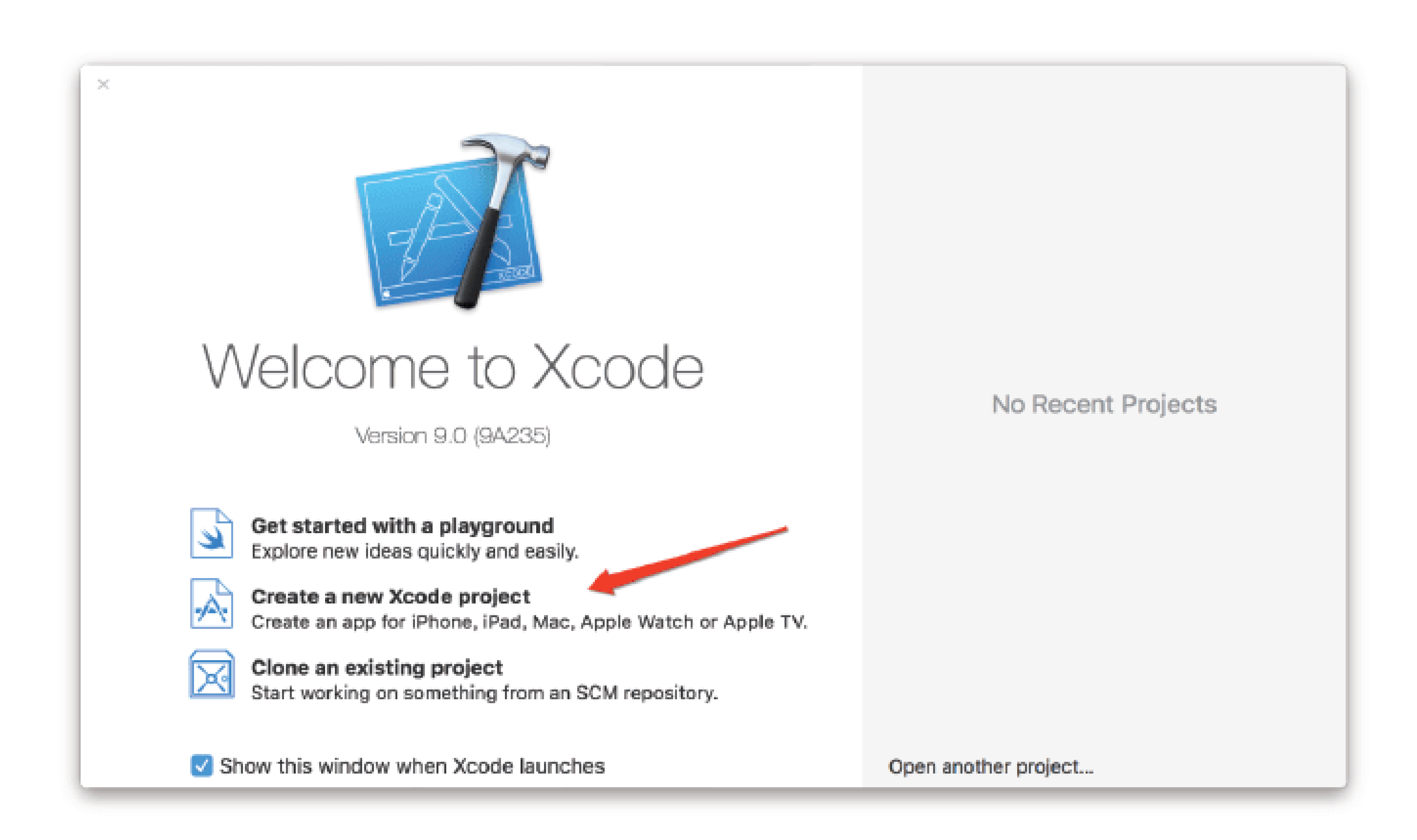


图 12-5 选择 Create a new Xcode project 创建工程

步骤 02 在弹出的对话框中选择 Single View App，然后单击 Next 按钮，如图 12-6 所示。

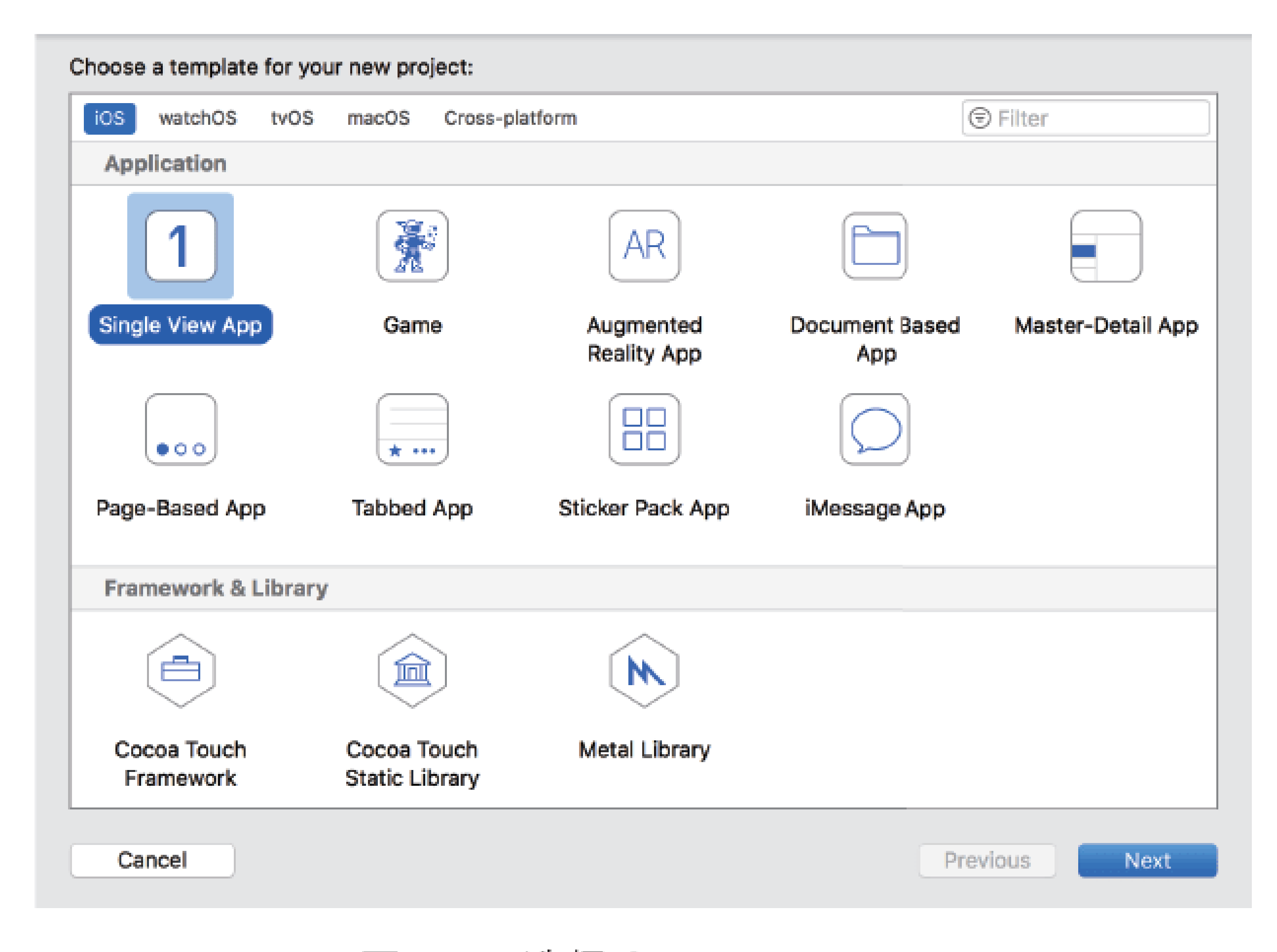


图 12-6 选择 Single View App

步骤 03　在弹出的对话框中输入工程名（如 CAM），然后选择使用 Objective-C 语言进行开发，因为要使用 OpenCV，它暂时不支持 Swift，如图 12-7 所示。

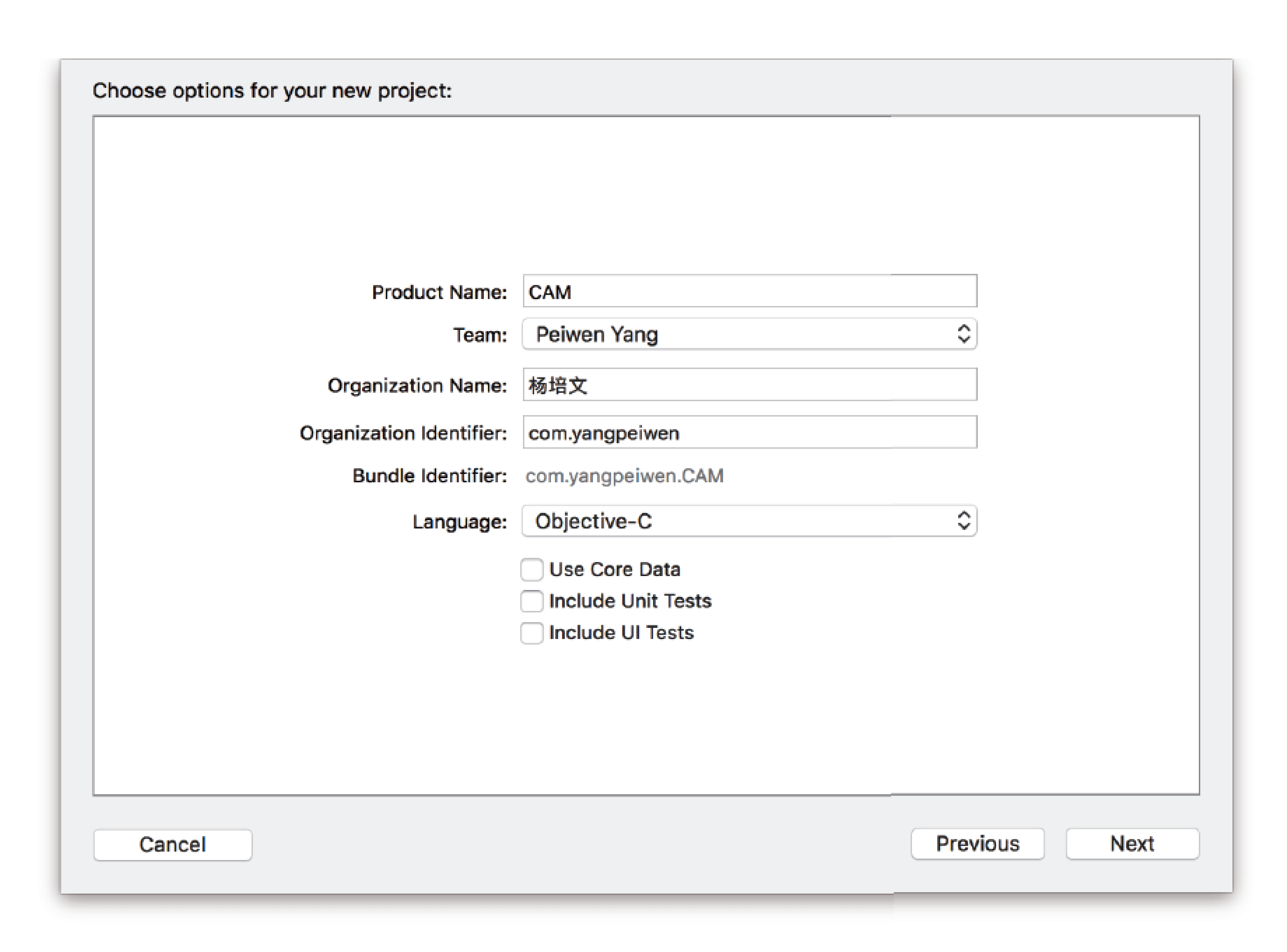


图 12-7　输入工程名和选择语言

步骤 04　在弹出的对话框中，关闭横屏模式，如图 12-8 所示。

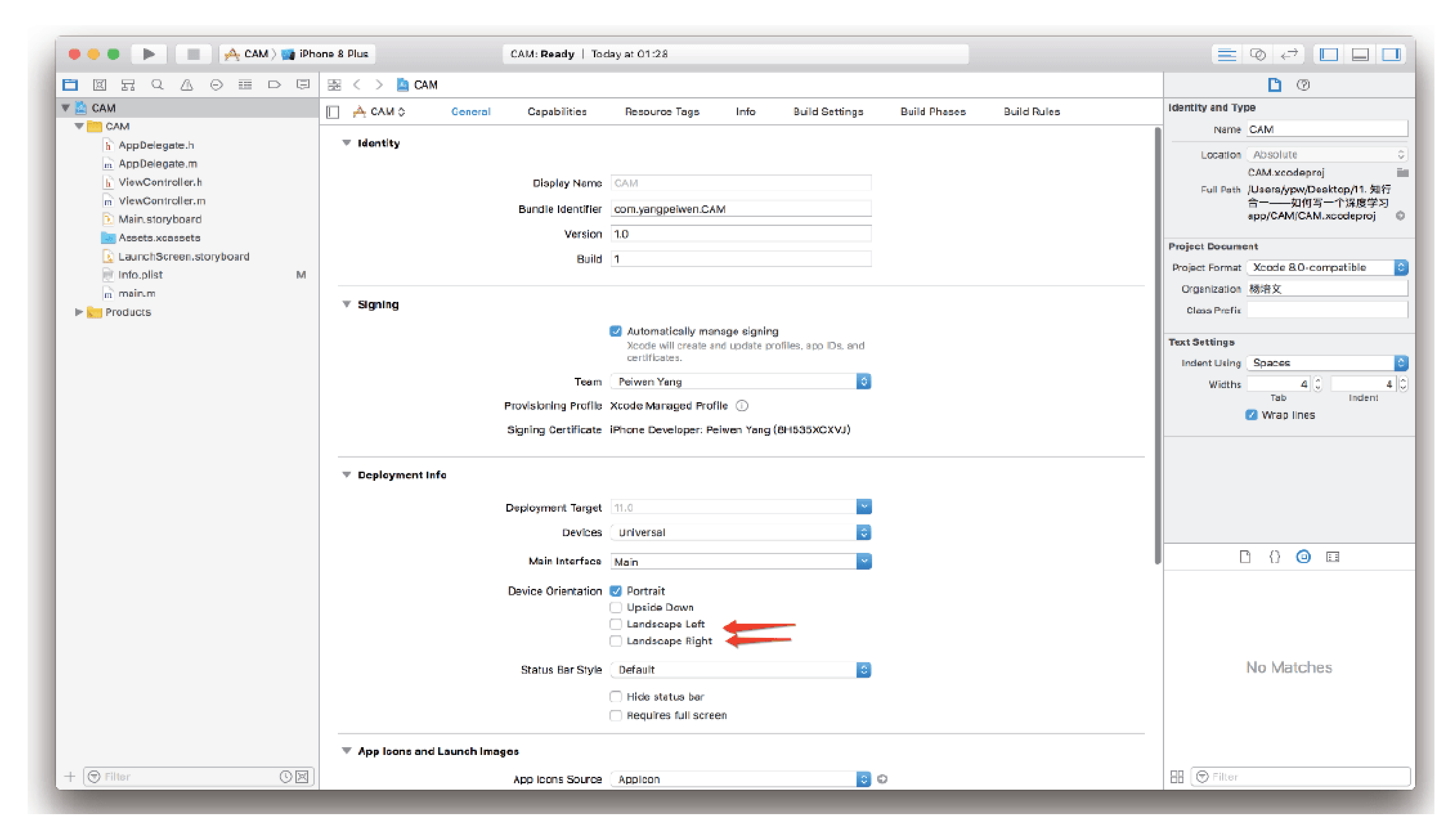


图 12-8　关闭横屏模式

12.3.2 配置工程

接下来开始配置工程，需要添加库文件、添加依赖项、在Info中添加摄像头权限、修改后缀名、添加必要的头文件、配置好委托事件以及摄像头等。

1. 添加库文件

添加库文件的步骤如下：

步骤 01 首先下载 OpenCV 为 iOS 系统编译的库文件，然后拖入 OpenCV（opencv2.framework）到工程左边的文件列表中，如图 12-9 所示。

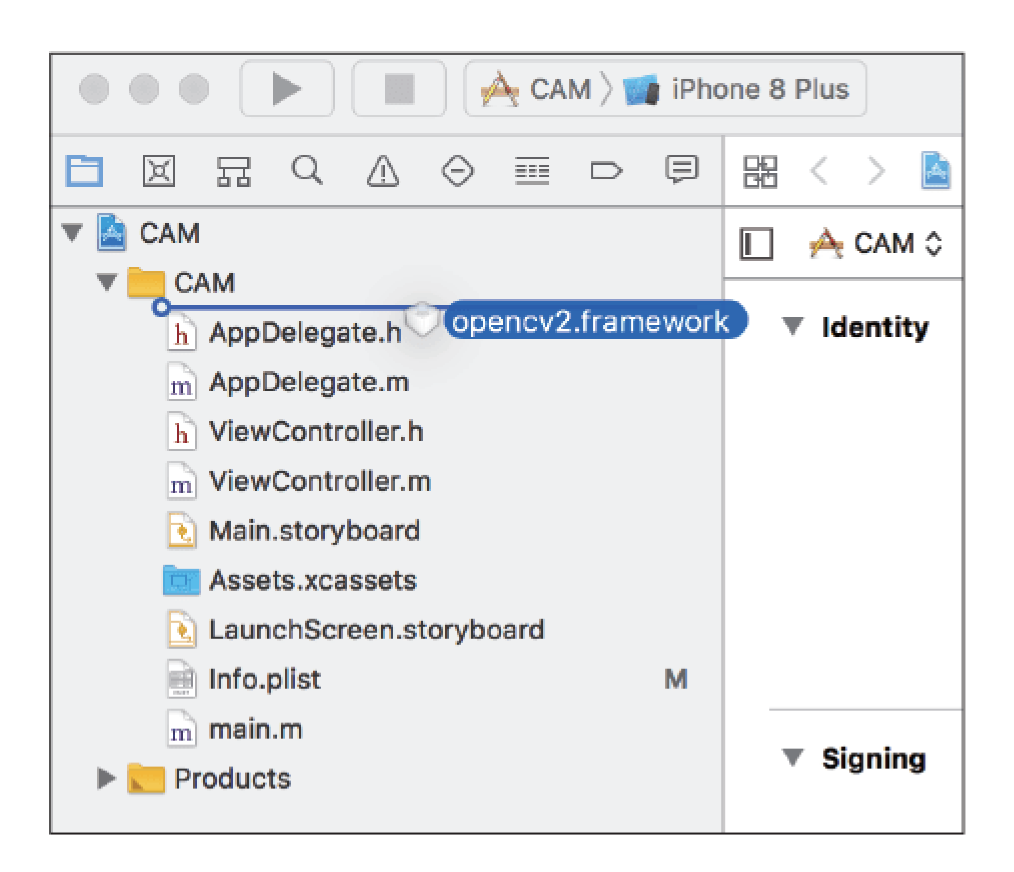


图 12-9 将 opencv2.framework 拖入左侧列表中

步骤 02 单击 Finish 按钮完成添加，如图 12-10 所示。

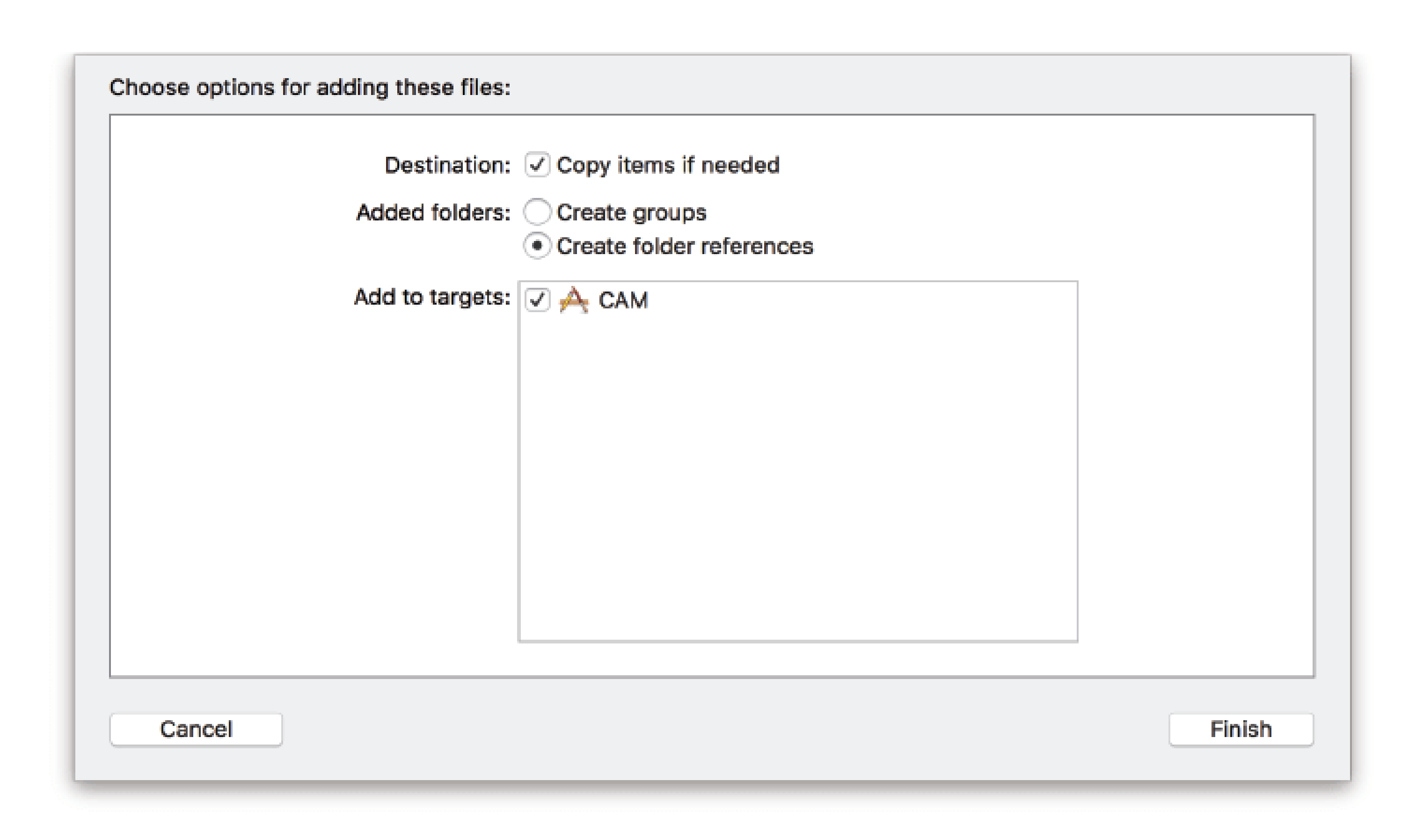


图 12-10 单击 Finish 完成添加

2. 添加依赖库

因为 OpenCV 调用摄像头、进行流媒体处理需要一些库，所以需要添加依赖库，可以在 CAM → Build Phases → Link Binary With Libraries 选项中配置依赖的库，如图 12-11 所示。

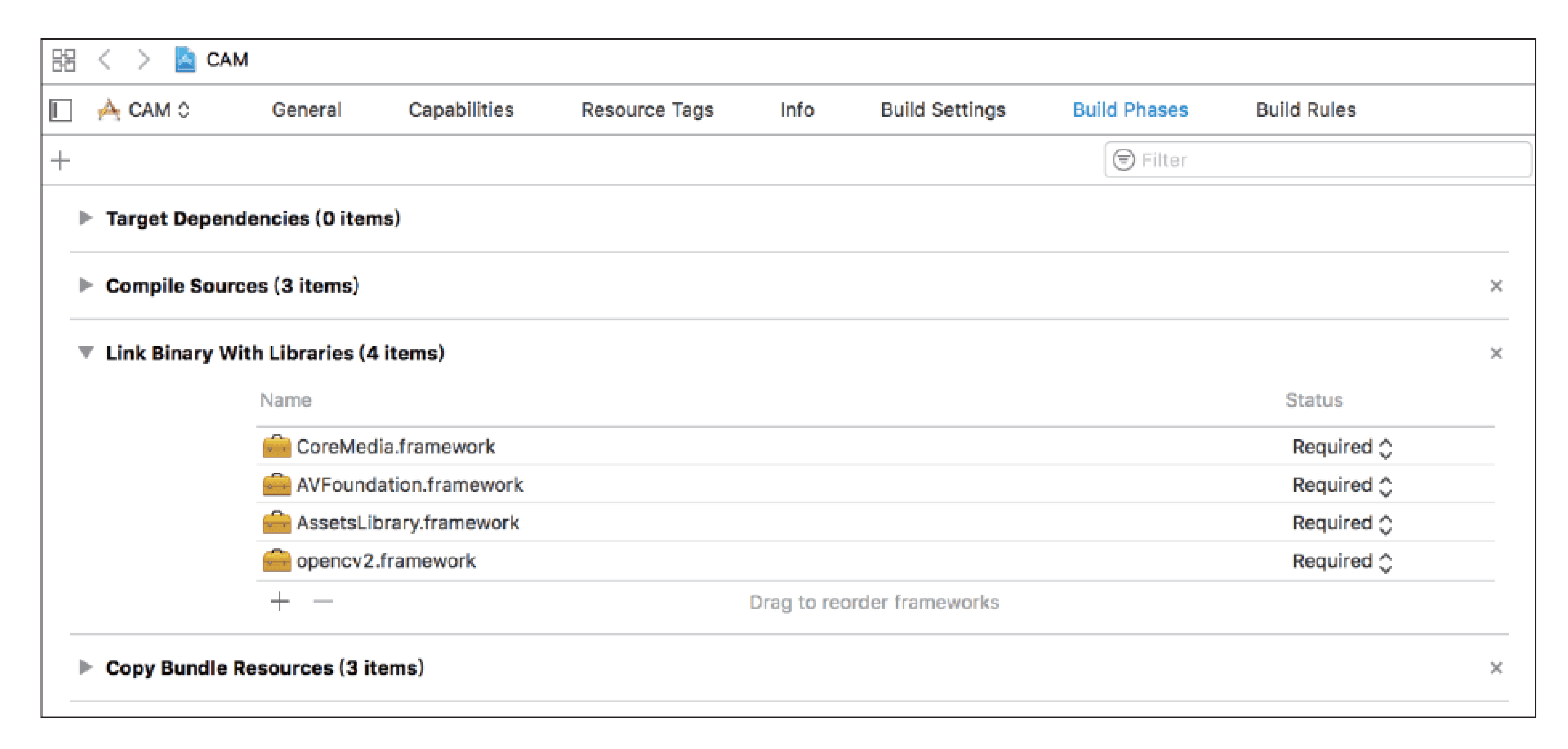


图 12-11　选择要添加的依赖库

主要添加以下依赖库：

- AssetsLibrary
- AVFoundation
- CoreMedia

3. 在 Info 中添加摄像头权限

因为本应用目的是通过摄像头分辨猫狗，iOS 调用摄像头需要经过用户同意，为了弹出用户同意的窗口，需要添加摄像头权限，在 Info 中的 Supported interface orientations 栏下添加 Privacy - Camera Usage Description 选项，然后写上提示语，如图 12-12 所示。

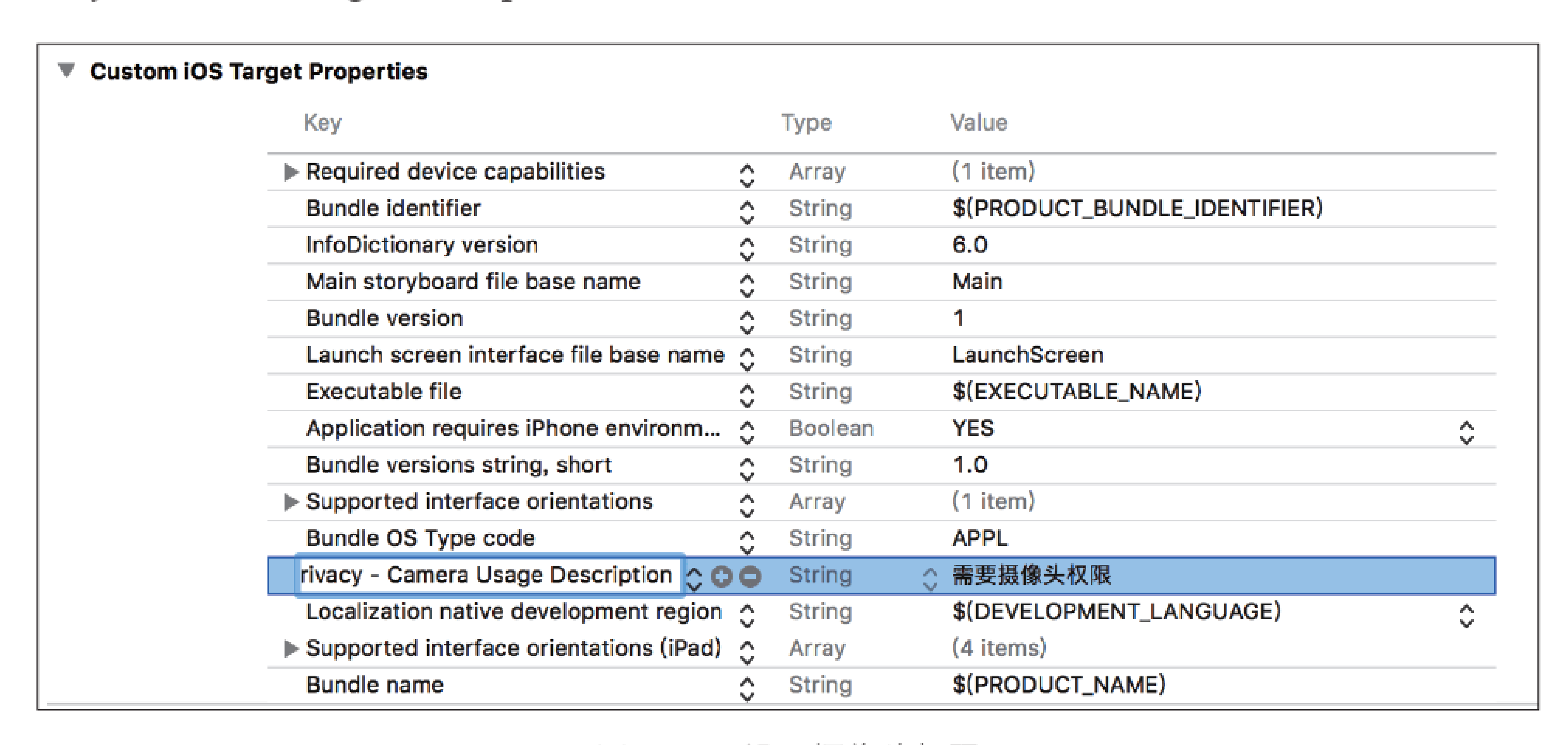


图 12-12　设置摄像头权限

4. 修改后缀名

为了支持 C++，需要将 ViewController.m 文件的后缀名修改为 ViewController.mm，如图 12-13 所示。

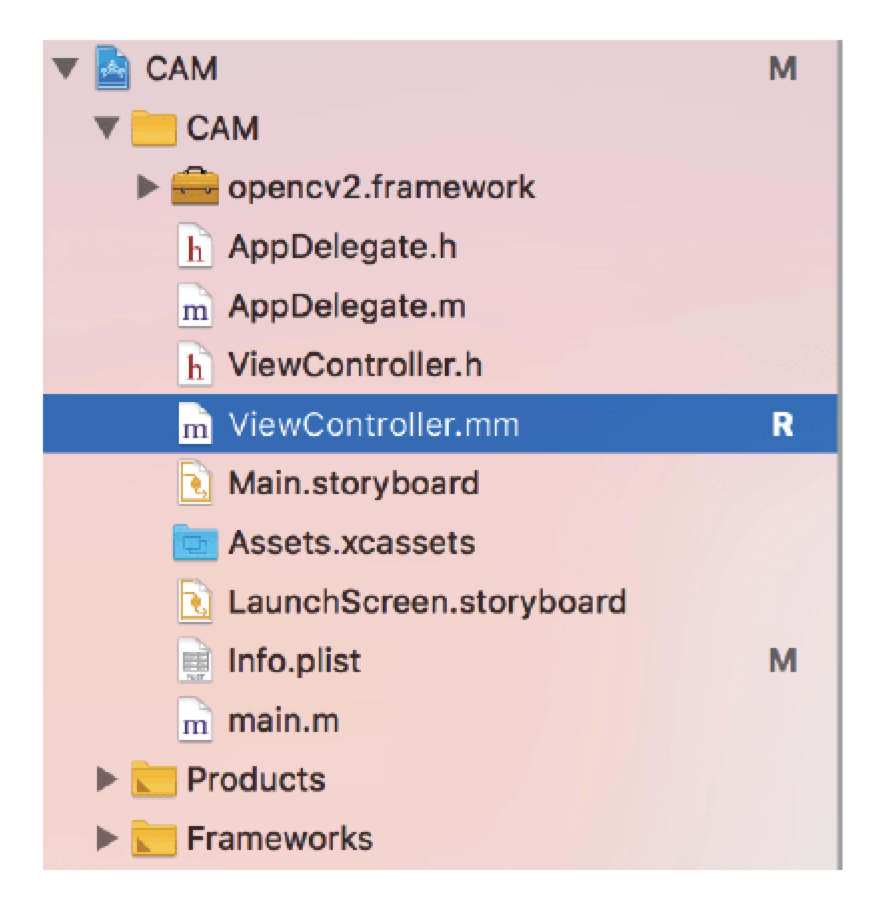


图 12-13 修改后缀名

5. 添加必要的头文件

在 ViewController.h 文件中添加必要的头文件：

```
#import <opencv2/videoio/cap_ios.h>
#import <opencv2/imgcodecs/ios.h>
#import <opencv2/highgui/highgui.hpp>
#import <opencv2/imgproc/imgproc.hpp>
```

6. 配置好委托事件以及摄像头

首先在 ViewController.h 文件中的 UIViewController 后面添加一个委托 <CvVideoCameraDelegate>，如图 12-14 所示。

```
//
//  ViewController.h
//  CAM
//
//  Created by 杨培文 on 2017/10/4.
//  Copyright © 2017年 杨培文. All rights reserved.
//

#import <UIKit/UIKit.h>
#import <opencv2/videoio/cap_ios.h>
#import <opencv2/imgcodecs/ios.h>
#import <opencv2/highgui/highgui.hpp>
#import <opencv2/imgproc/imgproc.hpp>

@interface ViewController : UIViewController <CvVideoCameraDelegate>


@end

```

图 12-14 添加委托 <CvVideoCameraDelegate>

然后在 ViewController.mm 中添加函数：

```
- (void)processImage:(cv::Mat &)input_img {

}
```

当摄像头有新的图像传过来的时候，就会调用这个函数来传入图片，可以在这个函数中编写图像处理程序。

我们还需要在 viewDidLoad 函数中添加初始化摄像头的代码：

```
    CvVideoCamera * camera;
    - (void)viewDidLoad {
        [super viewDidLoad];
        // 初始化摄像头
        camera = [[CvVideoCamera alloc] init];
        camera.defaultAVCaptureDevicePosition = AVCaptureDevicePositionBack;
        camera.defaultAVCaptureSessionPreset = AVCaptureSessionPreset192
0x1080;
        camera.defaultAVCaptureVideoOrientation = AVCaptureVideoOrientationP
ortrait;
        camera.defaultFPS = 30;
        camera.grayscaleMode = false;
        camera.delegate = self;

        [camera start];
    }
```

7. 添加 UIImageView 到 Main.storyboard

为了能够显示图片，需要在 Main.storyboard 中添加一个 UIImageView，然后调整它的位置，建议放置在如图 12-15 所示的位置。

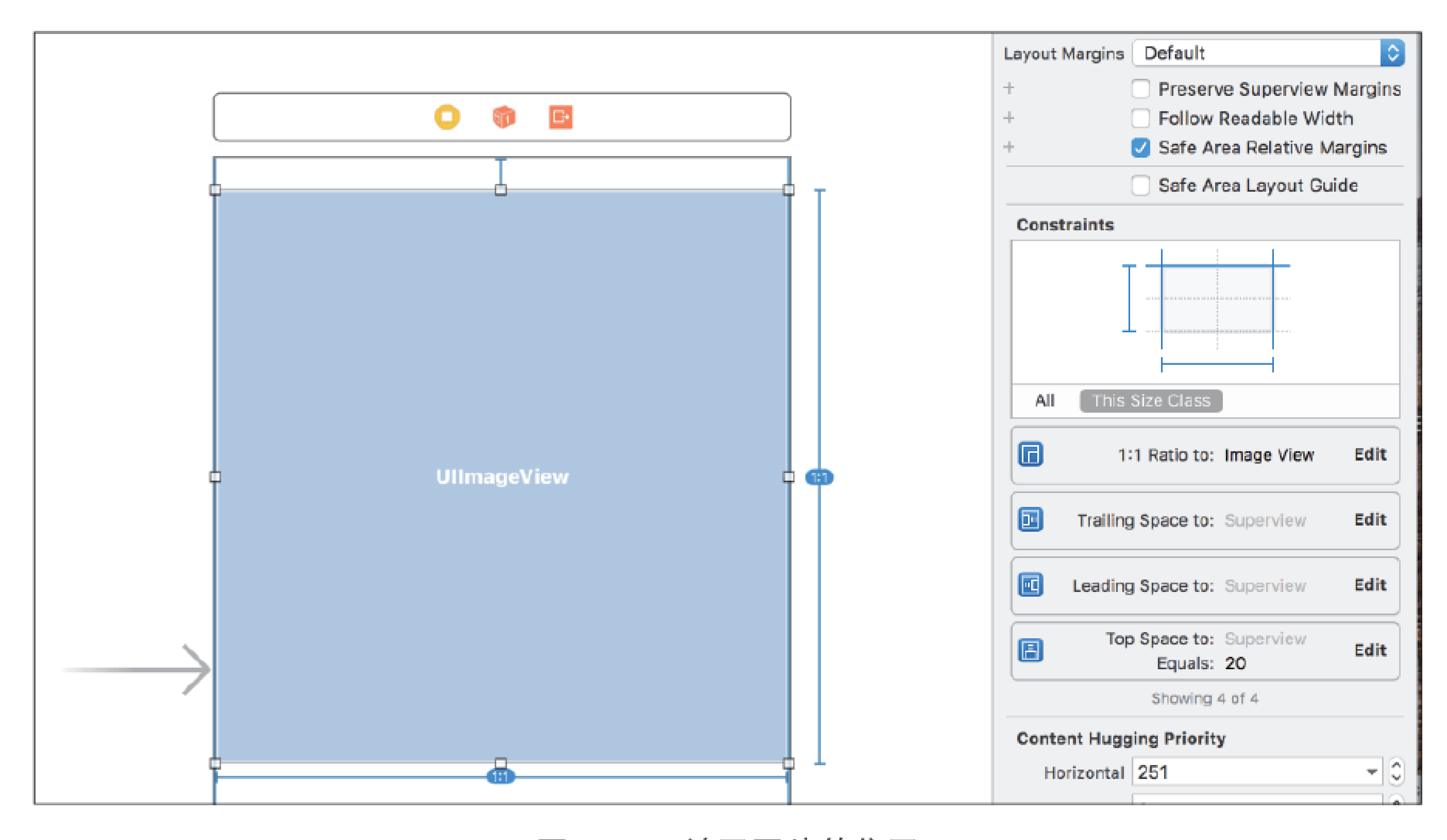


图 12-15　放置图片的位置

设置图片的宽高比为 1:1，以显示模型输入图片的效果，距离顶部一定的距离，避免时间显示在图片上。

接下来需要将 UIImageView 链接到程序中，首先单击右上角的两个圆圈（Show the Assistant editor），切换到如图 12-16 所示的界面，然后按住 Ctrl 键将 UIImageView 拖到如图 12-17 所示的位置。

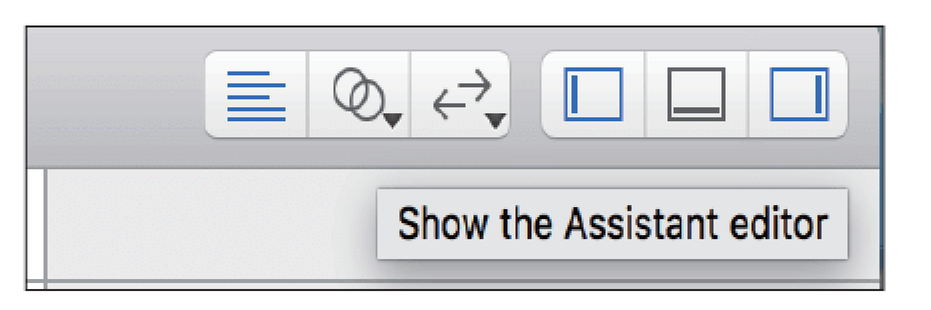


图 12-16 打开辅助编辑器

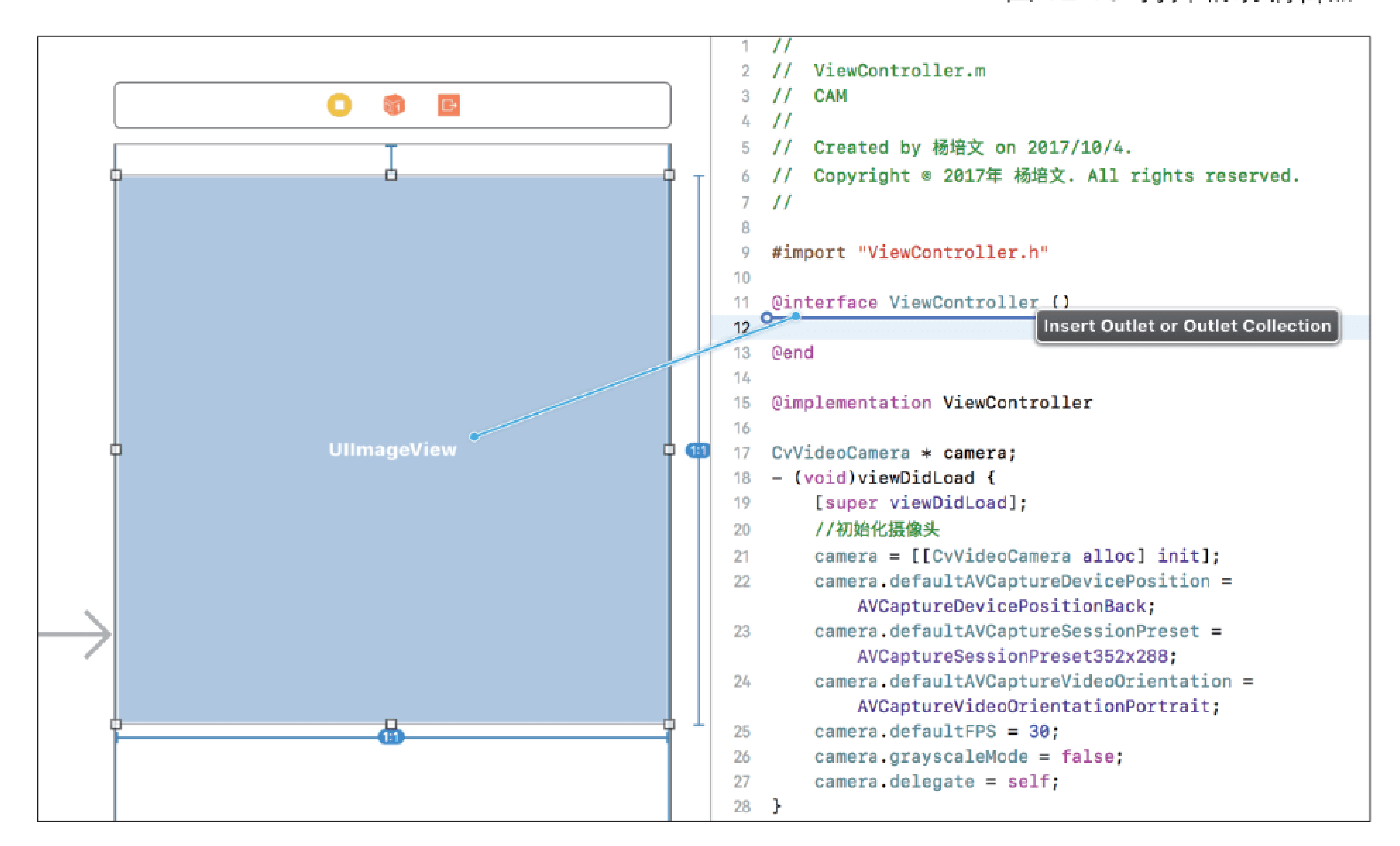


图 12-17 拖动 UIImageView 到相应的位置

接下来输入名称 imageView，就成功将 UI 链接到代码中了，如图 12-18 所示。

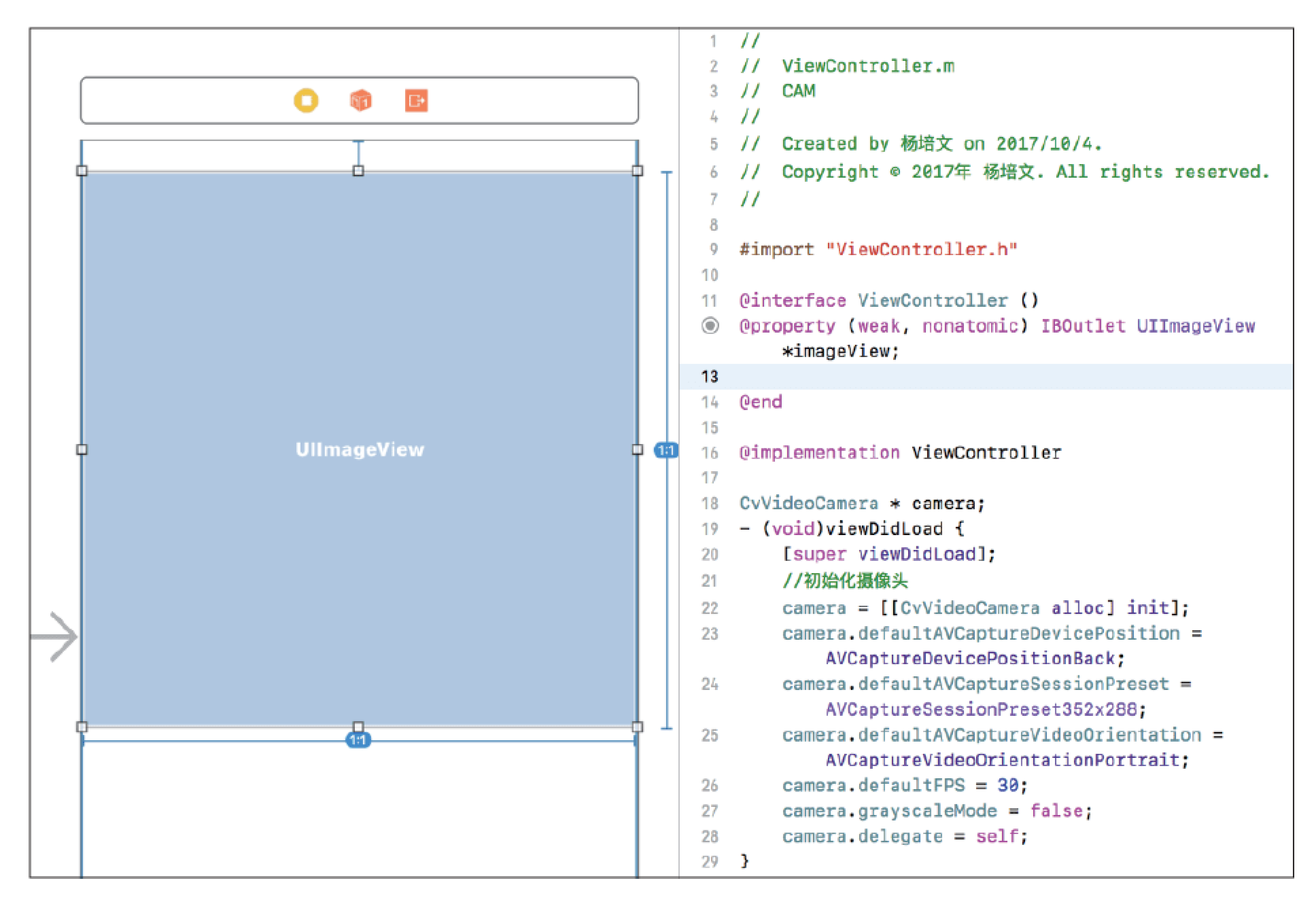


图 12-18 输入名称 imageView

12.3.3 测试工程

我们可以先做一个简单的读取摄像头 App，在 Main.storyboard 中放置一个 ImageView，用于显示摄像头图片，然后在 processImage 函数中编写显示图片的代码：

```
- (void)processImage:(cv::Mat &)input_img {
    cv::cvtColor(input_img, input_img, CV_BGR2RGB);
    dispatch_async(dispatch_get_main_queue(), ^{
        _imageView.image = MatToUIImage(input_img);
    });
}
```

第一句代码是因为 OpenCV 的图像格式是 BGR，iOS 的顺序是 RGB，因此需要进行转换，第二句意思是显示代码需要在主线程中执行，不然 UI 不会更新。

12.3.4 运行程序

将手机与电脑连接，设置调试设备为你的手机（可以看到图中的手机叫 YPW），然后单击 ▶ 按钮开始运行，测试项目，如图 12-19 所示。

可以在手机上看到摄像头拍摄到的画面，如图 12-20 所示。

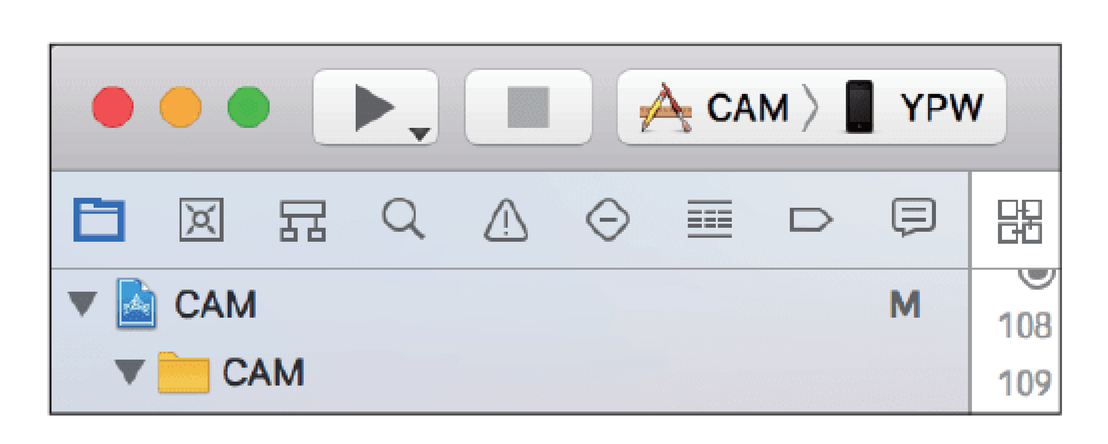


图 12-19 将手机与电脑连接

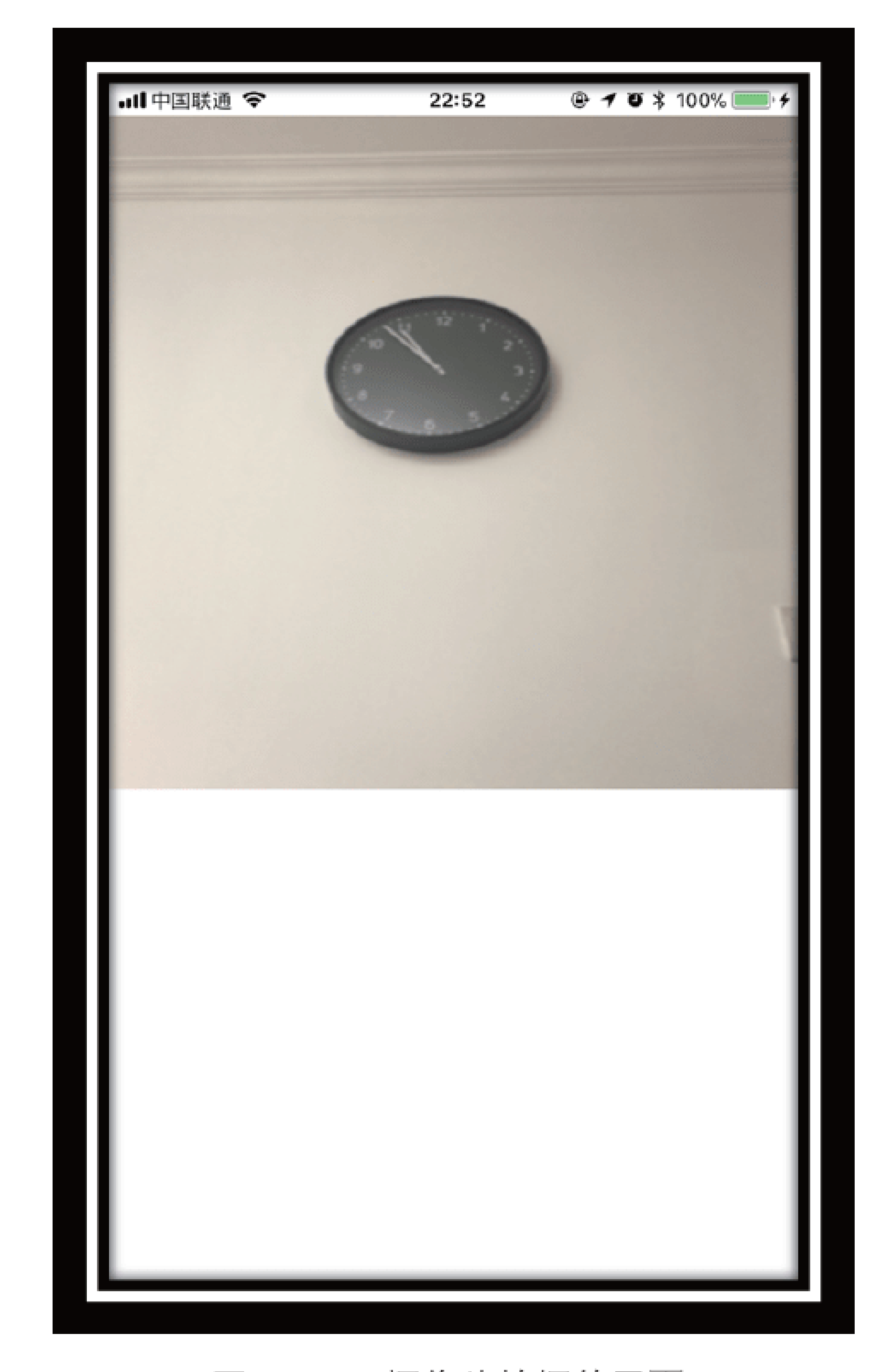


图 12-20 摄像头拍摄的画面

12.4 使用深度学习模型

本节会在 12.3 节搭建工程的基础上，使用之前导出的模型对摄像头拍到的图像进行猫狗分类，并且进行 CAM 可视化。之前模型导出部分，详见本书附带的代码 https://github.com/Jinglue/DL4Img/blob/master/notebook/Lecture12.ipynb。

12.4.1 将模型导入到工程中

将模型导入到工程中的操作步骤如下：

步骤 01 将模型拖入左侧的工程文件夹中，然后单击模型文件，配置模型的 Target Membership 为 CAM，如图 12-21 所示。

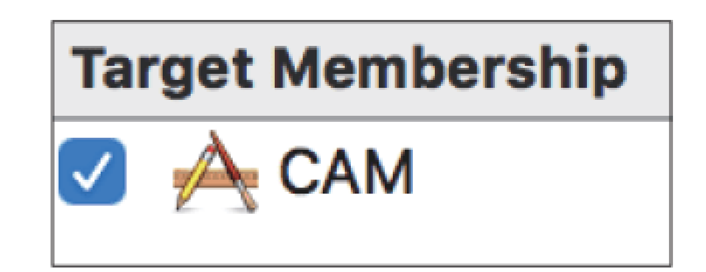


图 12-21 将模型添加到项目中

步骤 02 在头文件中引入模型文件：

```
#import "model_clf.h"
#import "model_cam.h"
```

如果这一步报错，请检查模型是否正确导入到工程中。

12.4.2 数据类型转换函数

由于我们的图片是通过 OpenCV 调用摄像头获取的，获取到的图片格式是 cv::Mat，但是模型的输入类型为 CVPixelBufferRef，因此需要一个转换函数：

```
- (CVPixelBufferRef)pixelBufferFromCGImage:(CGImageRef)image
{
    NSDictionary *options = @{(id)kCVPixelBufferCGImageCompatibilityKey: @YES,
                (id)kCVPixelBufferCGBitmapContextCompatibilityKey: @YES};
    CVPixelBufferRef pxbuffer = NULL;
size_t width = CGImageGetWidth(image);
height = CGImageGetHeight(image);
CVReturn status = CVPixelBufferCreate(kCFAllocatorDefault,
                   width, height, kCVPixelFormatType_32ARGB,
                   (__bridge CFDictionaryRef) options, &pxbuffer
  );
    NSParameterAssert(status == kCVReturnSuccess && pxbuffer != NULL);

    CVPixelBufferLockBaseAddress(pxbuffer, 0);
    void *pxdata = CVPixelBufferGetBaseAddress(pxbuffer);
    NSParameterAssert(pxdata != NULL);

    CGColorSpaceRef rgbColorSpace = CGColorSpaceCreateDeviceRGB();
CGContextRef context = CGBitmapContextCreate(
```

```
            pxdata, width, height, 8, 4*width,
            rgbColorSpace, kCGImageAlphaNoneSkipFirst
    );
     NSParameterAssert(context);

     CGContextDrawImage(context, CGRectMake(0, 0, width, height), image);
     CGColorSpaceRelease(rgbColorSpace);
     CGContextRelease(context);

     CVPixelBufferUnlockBaseAddress(pxbuffer, 0);

     return pxbuffer;
 }
```

这个函数可以将 CGImage 转换为 CVPixelBufferRef，因此图像转换全过程为：先通过 processImage 函数获取到 cv::Mat 格式的图片，再通过 OpenCV 自带的 MatToUIImage 函数转换为 UIImage 类型，UIImage 格式很容易转换为 CGImage 格式，调用方法是 image.CGImage，然后传入 pixelBufferFromCGImage 得到 CVPixelBufferRef 格式。

1. 转换图像为正方形

由于模型有全连接，因此必须输入固定尺寸的图像。固定尺寸有两种方式，一种是切割再缩放，另一种是直接拉伸。直接拉伸会改变宽高比，从 16:9 拉到 1:1 时改变比较大，所以还是保持宽高比好一些，选择先切割再缩放的方法。

```
// 转换图像为正方形
cv::cvtColor(input_img, input_img, CV_BGR2RGB);
input_img = input_img(cv::Rect(0, 0, input_img.cols, input_img.cols));
cv::Mat smallImage;
cv::resize(input_img, smallImage, cv::Size(224, 224));
UIImage * image = MatToUIImage(smallImage);
```

2. 用模型进行预测

转换为（224, 224, 3）的图片以后，就可以输入到模型进行预测了。

首先将 UIImage 转换为 CVPixelBufferRef，然后直接调用模型的 predictionFromImage 函数。对于分类器（clf）预测出来的结果，直接取第一个值就好。CAM 模型的输出是一个矩阵，还需要进一步转换才能使用 OpenCV 的函数进行处理。

代码如下：

```
        CVPixelBufferRef bufferRef = [self pixelBufferFromCGImage:image.CGImage];
        float prediction = [clf predictionFromImage:bufferRef error:nil].
prediction[0].floatValue;
```

```
        MLMultiArray * featuremap = [cam predictionFromImage:bufferRef
error:nil].cam;
        CVPixelBufferRelease(bufferRef);
```

12.4.3 实施 CAM 可视化

接下来就是实现上面可视化测试的 C++ 版。首先转换 MLMultiArray 类型为 cv::Mat 类型，然后调整范围，染色，并加在原图上，生成 CAM 可视化图。染色是从 0~1 映射到蓝 ~ 红的一个过程。

1. 将 MLMultiArray 转换为 Mat

首先构建一个（7, 7）的图，然后从 featuremap 中取数。可以从模型中看到，模型的输出 shape 是 (2, 7, 7)，如图 12-22 所示。

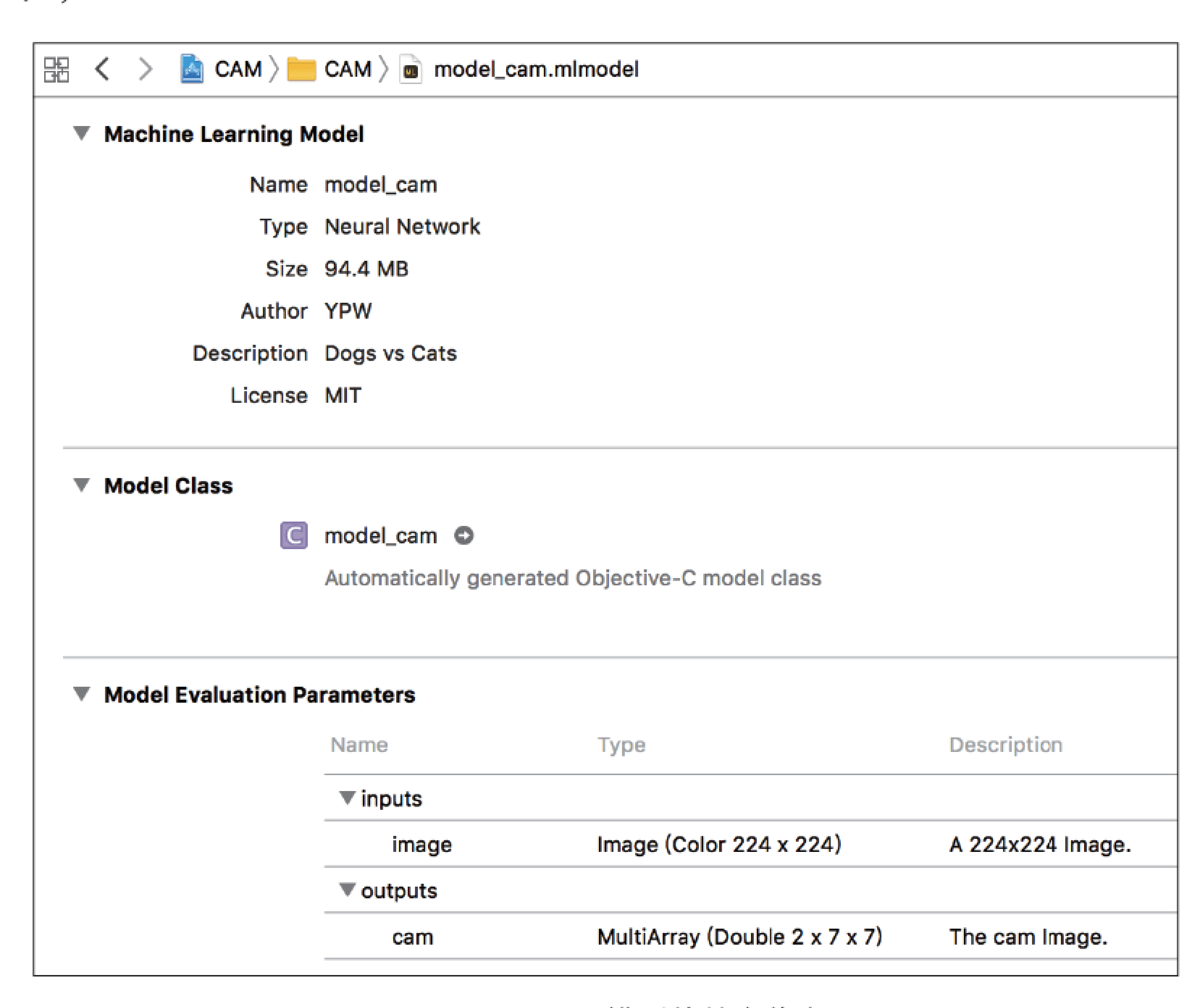


图 12-22 CAM 模型的基本信息

因此，可以写出这样的代码：如果是猫，就取第一个 CAM 图，下标是 [i*7+j]，否则取第二个，下标是 [49+i*7+j]。

```
    cv::Mat img_cam(7, 7, CV_32F);
    for(int i = 0; i < 7; i++)
    {
        for(int j = 0; j < 7; j++){
            if(prediction > 0.5){
                img_cam.row(i).col(j) = featuremap[i*7+j].floatValue;
```

```
            }
            else
            {
                img_cam.row(i).col(j) = featuremap[49+i*7+j].floatValue;
            }
        }
    }
```

2. 调整 cam 的范围

此部分与之前的 Python 代码是一个逻辑，先缩放，再限制在 0 ～ 1 的范围内，放大到和原图一样的尺寸，并转换成 uint8 格式。

有一点不同的就是，缩放的尺寸不是 (224, 224)，而是 (1080, 1080)，目的是为了让图片更加清晰。

```
int width = input_img.rows;
img_cam /= 10;
img_cam.setTo(0, img_cam < 0);
img_cam.setTo(1, img_cam > 1);
cv::resize(img_cam, img_cam, cv::Size(width, width));

cv::Mat img_cam2;
img_cam2 = img_cam * 255;
img_cam2.convertTo(img_cam2, CV_8U);
```

3. 染成彩色和加在原图上

这部分代码的逻辑也与之前的 Python 代码一样：

```
// 染成彩色
cv::Mat heatmap(width, width, CV_8UC3);
cv::applyColorMap(img_cam2, heatmap, cv::COLORMAP_JET);
cv::cvtColor(heatmap, heatmap, CV_BGR2RGB);

// 加在原图上
cv::Mat outImage(width, width, CV_8UC3);
cv::addWeighted(input_img, 0.8, heatmap, 0.4, 0, outImage);
```

4. 显示图片和文字

最后将概率显示在 Label 上，将图片显示在 imageView 上。这里需要在 Main.storyboard 中拖出一个 Label 到界面中央，然后链接到代码中。右侧可以看到约束条件，如图 12-23 所示。

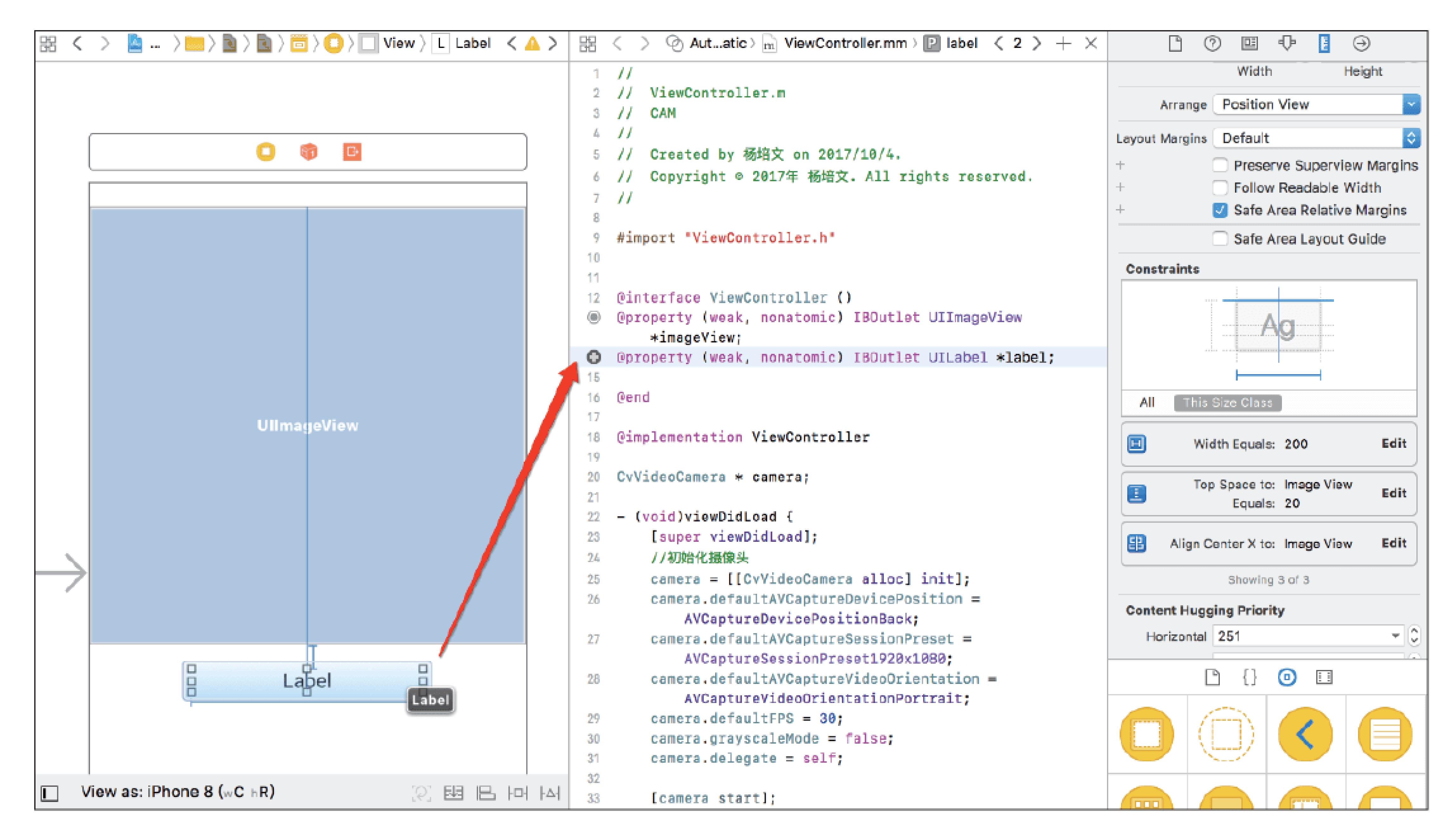


图 12-23 添加一个 Label

```
    NSString * resultString;
    if(prediction < 0.5){
        resultString = [NSString stringWithFormat:@"狗的概率：%.2f",
1-prediction];
    }
    else{
        resultString = [NSString stringWithFormat:@"猫的概率：%.2f",
prediction];
    }
    dispatch_async(dispatch_get_main_queue(), ^{
        _label.text = resultString;
        _imageView.image = MatToUIImage(outImage);
    });
```

12.4.4 模型效果

我们可以看到模型通过猫的脸、猫的胡子以及猫的尾巴来判断这是一只猫的概率为 0.99，效果很棒，如图 12-24 所示。

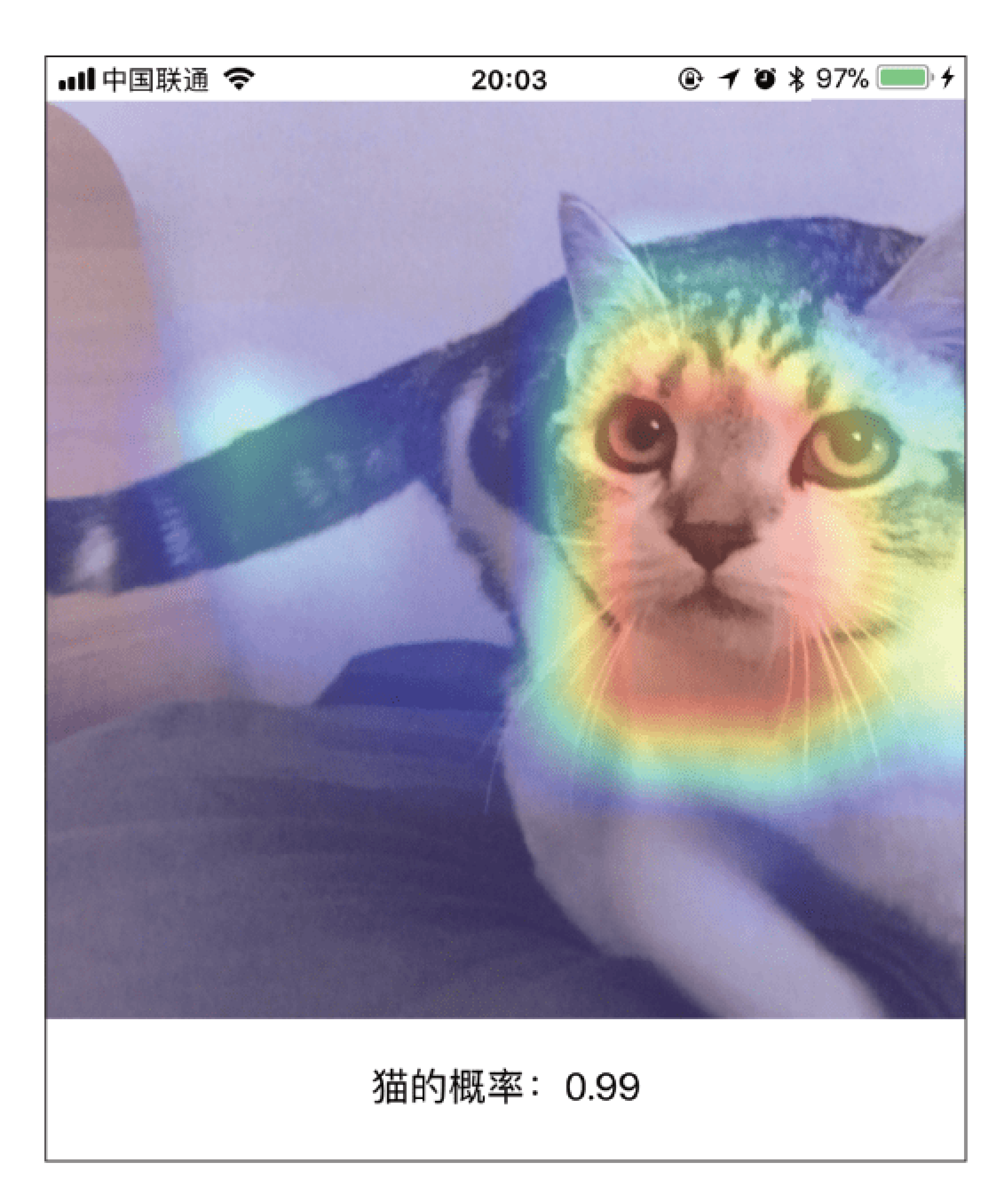


图 12-24　显示猫的效果

如果有需要，还可以继续编写暂停按钮、拍照按钮、画出概率曲线等功能。

12.5 参考文献及网页链接

［1］ OpenCV: ColorMaps in OpenCV. Available at: http://docs.opencv.org/trunk/d3/d50/group__imgproc__colormap.html.

［2］ Converting Trained Models to Core ML. Converting Trained Models to Core ML | Apple Developer Documentation. Available at: https://developer.apple.com/documentation/coreml/converting_trained_models_to_core_ml.

［3］ Learning Deep Features for Discriminative Localization. CNN Discriminative Localization and Saliency - MIT. Available at: http://cnnlocalization.csail.mit.edu/.

［4］ Using AVAssetWriter to create a movie from images is not working as expected on a 3GS device. iphone - Using AVAssetWriter to create a movie from images is not working as expected on a 3GS device - Stack Overflow. Available at: https://stackoverflow.com/questions/5681927/using-avassetwriter-to-create-a-movie-from-images-is-not-working-as-expected-on.

［5］ Ypwhs. ypwhs/dogs_vs_cats. GitHub. Available at: https://github.com/ypwhs/dogs_vs_cats/blob/master/transfer_learning.ipynb.